Microsoft

Developing SQL Databases

Exam Ref 70-762

Louis Davidson

Exam Ref 70-762 Developing SQL Databases

Louis Davidson
Stacia Varga

Exam Ref 70-762 Developing SQL Databases

Published with the authorization of Microsoft Corporation by:
Pearson Education, Inc.

Copyright © 2017 by Pearson Education Inc.

All rights reserved. Printed in the United States of America. This publication is protected by copyright, and permission must be obtained from the publisher prior to any prohibited reproduction, storage in a retrieval system, or transmission in any form or by any means, electronic, mechanical, photocopying, recording, or likewise. For information regarding permissions, request forms, and the appropriate contacts within the Pearson Education Global Rights & Permissions Department, please visit www.pearsoned.com/permissions/. No patent liability is assumed with respect to the use of the information contained herein. Although every precaution has been taken in the preparation of this book, the publisher and author assume no responsibility for errors or omissions. Nor is any liability assumed for damages resulting from the use of the information contained herein.

ISBN-13: 978-1-5093-0491-2
ISBN-10: 1-5093-0491-6

Library of Congress Control Number: 2016962647

First Printing January 2017

Trademarks

Microsoft and the trademarks listed at https://www.microsoft.com on the "Trademarks" webpage are trademarks of the Microsoft group of companies. All other marks are property of their respective owners.

Warning and Disclaimer

Every effort has been made to make this book as complete and as accurate as possible, but no warranty or fitness is implied. The information provided is on an "as is" basis. The authors, the publisher, and Microsoft Corporation shall have neither liability nor responsibility to any person or entity with respect to any loss or damages arising from the information contained in this book or programs accompanying it.

Special Sales

For information about buying this title in bulk quantities, or for special sales opportunities (which may include electronic versions; custom cover designs; and content particular to your business, training goals, marketing focus, or branding interests), please contact our corporate sales department at corpsales@pearsoned.com or (800) 382-3419.

For government sales inquiries, please contact governmentsales@pearsoned.com.

For questions about sales outside the U.S., please contact intlcs@pearson.com.

Editor-in-Chief
Greg Wiegand

Acquisitions Editor
Trina MacDonald

Development Editor
Rick Kughen

Managing Editor
Sandra Schroeder

Senior Project Editor
Tracey Croom

Editorial Production
Backstop Media

Copy Editor
Jordan Severns

Indexer
Julie Grady

Proofreader
Christina Rudloff

Technical Editor
Christopher Ford

Cover Designer
Twist Creative, Seattle

Contents at a glance

Introduction

Preparing for the exam

CHAPTER 1 Design and implement database objects

CHAPTER 2 Implement programmability objects

CHAPTER 3 Manage database concurrency

CHAPTER 4 Optimize database objects and SQL infrastructure

Index

Contents

Introduction

Organization of this book

Microsoft certifications

Acknowledgments

Free ebooks from Microsoft Press

Microsoft Virtual Academy

Quick access to online references

Errata, updates, & book support

We want to hear from you

Stay in touch

Preparing for the exam

Chapter 1 Design and implement database objects

Skill 1.1: Design and implement a relational database schema

Designing tables and schemas based on business requirements

Improving the design of tables by using normalization

Writing table create statements

Determining the most efficient data types to use

Skill 1.2: Design and implement indexes

Design new indexes based on provided tables, queries, or plans

Distinguish between indexed columns and included columns

Implement clustered index columns by using best practices

Recommend new indexes based on query plans

Skill 1.3: Design and implement views

Design a view structure to select data based on user or business requirements

Identify the steps necessary to design an updateable view

Implement partitioned views

Implement indexed views

Skill 1.4: Implement columnstore indexes

Determine use cases that support the use of columnstore indexes

Identify proper usage of clustered and non-clustered columnstore indexes

Design standard non-clustered indexes in conjunction with clustered columnstore indexes

Implement columnstore index maintenance

Summary

Thought experiment

Thought experiment answer

Chapter 2 Implement programmability objects

Skill 2.1 Ensure data integrity with constraints

Define table and foreign-key constraints to enforce business rules

Write Transact-SQL statements to add constraints to tables

Identify results of Data Manipulation Language (DML) statements given existing tables and constraints

Identify proper usage of PRIMARY KEY constraints

Skill 2.2 Create stored procedures

Design stored procedure components and structure based on business requirements

Implement input and output parameters

Implement table-valued parameters

Implement return codes

Streamline existing stored procedure logic

Implement error handling and transaction control logic within stored procedures

Skill 2.3 Create triggers and user-defined functions

Design trigger logic based on business requirements

Determine when to use Data Manipulation Language (DML) triggers, Data Definition Language (DDL) triggers, or logon triggers

Recognize results based on execution of AFTER or INSTEAD OF triggers

Design scalar-valued and table-valued user-defined functions based on business requirements

Identify differences between deterministic and non-deterministic functions

Summary

Thought Experiment

Though Experiment Answer

Chapter 3 Manage database concurrency

Skill 3.1: Implement transactions
Identify DML statement results based on transaction behavior
Recognize differences between and identify usage of explicit and implicit transactions
Implement savepoints within transactions
Determine the role of transactions in high-concurrency databases
Skill 3.2: Manage isolation levels
Identify differences between isolation levels
Define results of concurrent queries based on isolation level
Identify the resource and performance impact of given isolation levels
Skill 3.3: Optimize concurrency and locking behavior
Troubleshoot locking issues
Identify lock escalation behaviors
Capture and analyze deadlock graphs
Identify ways to remediate deadlocks
Skill 3.4: Implement memory-optimized tables and native stored procedures
Define use cases for memory-optimized tables
Optimize performance of in-memory tables
Determine best case usage scenarios for natively compiled stored procedures
Enable collection of execution statistics for natively compiled stored procedures
Summary
Thought experiment
Thought experiment answers

Chapter 4 Optimize database objects and SQL infrastructure

Skill 4.1: Optimize statistics and indexes
Determine the accuracy of statistics and the associated impact to query plans and performance
Design statistics maintenance tasks
Use dynamic management objects to review current index usage and identify missing indexes
Consolidate overlapping indexes
Skill 4.2: Analyze and troubleshoot query plans
Capture query plans using extended events and traces

Identify poorly performing query plan operators
Compare estimated and actual query plans and related metadata
Configure Azure SQL Database Performance Insight
Skill 4.3: Manage performance for database instances
Manage database workload in SQL Server
Design and implement Elastic Scale for Azure SQL Database
Select an appropriate service tier or edition
Optimize database file and tempdb configuration
Optimize memory configuration
Monitor and diagnose schedule and wait statistics using dynamic management objects
Troubleshoot and analyze storage, IO, and cache issues
Monitor Azure SQL Database query plans
Skill 4.4: Monitor and trace SQL Server baseline performance metrics
Monitor operating system and SQL Server performance metrics
Compare baseline metrics to observed metrics while troubleshooting performance issues
Identify differences between performance monitoring and logging tools
Monitor Azure SQL Database performance
Determine best practice use cases for extended events
Distinguish between Extended Events targets
Compare the impact of Extended Events and SQL Trace
Define differences between Extended Events Packages, Targets, Actions, and Sessions
Chapter summary
Thought experiment
Thought experiment answer
Index

What do you think of this book? We want to hear from you!

Microsoft is interested in hearing your feedback so we can continually improve our books and learning resources for you. To participate in a brief online survey, please visit:

https://aka.ms/tellpress

Introduction

The 70-762 exam tests your knowledge about developing databases in Microsoft SQL Server 2016. To successfully pass this exam, you should know how to create various types of database objects, such as disk-based and memory-optimized tables, indexes, views, and stored procedures, to name a few. Not only must you know how and why to develop specific types of database objects, but you must understand how to manage database concurrency by correctly using transactions, assigning isolation levels, and troubleshooting locking behavior. Furthermore, you must demonstrate familiarity with techniques to optimize database performance by reviewing statistics and index usage, using tools to troubleshoot and optimize query plans, optimizing the configuration of SQL Server and server resources, and monitoring SQL Server performance metrics. You must also understand the similarities and differences between working with databases with SQL Server on-premises and Windows Azure SQL Database in the cloud.

The 70-762 exam is focused on measuring skills of database professionals, such as developers or administrators, who are responsible for designing, implementing, or optimizing relational databases by using SQL Server 2016 or SQL Database. In addition to reinforcing your existing skills, it measures what you know about new features and capabilities in SQL Server and SQL Database.

To help you prepare for this exam and reinforce the concepts that it tests, we provide many different examples that you can try for yourself. Some of these examples require only that you have installed SQL Server 2016 or have created a Windows Azure subscription. Other examples require that you download and restore a backup of the Wide World Importers sample database for SQL Server 2016 from *https://github.com/Microsoft/sql-server-samples/releases/tag/wide-world-importers-v1.0*. The file to download from this page is WideWorldImporters-Full.bak. You can find documentation about this sample database at Wide World Importers documentation, *https://msdn.microsoft.com/library/mt734199(v=sql.1).aspx*.

This book covers every major topic area found on the exam, but it does not cover every exam question. Only the Microsoft exam team has access to the exam questions, and Microsoft regularly adds new questions to the exam, making it impossible to cover specific questions. You should consider this book a supplement to your relevant real-world experience and other study materials. If you encounter a topic in this book that you do not feel completely comfortable with, use the "Need more review?" links you'll find in the text to find more information and take the time to research and study the topic. Great information is available on MSDN, TechNet, and in blogs and forums.

Organization of this book

This book is organized by the "Skills measured" list published for the exam. The "Skills

measured" list is available for each exam on the Microsoft Learning website: https://aka.ms/examlist. Each chapter in this book corresponds to a major topic area in the list, and the technical tasks in each topic area determine a chapter's organization. If an exam covers six major topic areas, for example, the book will contain six chapters.

Microsoft certifications

Microsoft certifications distinguish you by proving your command of a broad set of skills and experience with current Microsoft products and technologies. The exams and corresponding certifications are developed to validate your mastery of critical competencies as you design and develop, or implement and support, solutions with Microsoft products and technologies both on-premises and in the cloud. Certification brings a variety of benefits to the individual and to employers and organizations.

> **More Info All Microsoft certifications**
>
> For information about Microsoft certifications, including a full list of available certifications, go to https://www.microsoft.com/learning.

Acknowledgments

Louis Davidson I would like to dedicate my half of this book to my wife Valerie, who put up with me writing my half of this book (a few times) while simultaneously finishing my Database Design book.

Technically speaking, I would like to thank my colleagues in the MVP community and program at Microsoft. I have learned so much from them for the many years I have been an awardee and would never have accomplished so much without them. Far more than one is referenced for additional material.

Thank you, Stacia, for your work on the book. I appreciate your involvement more than you can imagine.

Stacia Varga I am grateful to have a community of SQL Server professionals that are always ready to share their experience and insights related with me, whether through informal conversations or more extensive reviews of any content that I write. The number of people with whom I have had informal conversations are too numerous to mention, but they know who they are. I would like to thank a few people in particular for the more in-depth help they provided: Joseph D'Antoni, Grant Fritchey, and Brandon Leach. And thanks to Louis as well. We have been on stage together, we have worked together, and now we have written together!

Behind the scenes of the publishing process, there are many other people involved that help us bring this book to fruition. I'd like to thank Trina McDonald for her role as the acquisitions editor and Troy Mott as the managing editor for his incredible patience with us and his efforts to make the process as easy as possible. I also appreciate the copyediting by

Christina Rudloff and technical editing by Christopher Ford to ensure that the information we provide in this book is communicated as clearly as possible and technically accurate.

Last, I want to thank my husband, Dean Varga, not only for tolerating my crazy work hours during the writing of this book, but also for doing his best to create an environment conducive to writing on many different levels.

Free ebooks from Microsoft Press

From technical overviews to in-depth information on special topics, the free ebooks from Microsoft Press cover a wide range of topics. These ebooks are available in PDF, EPUB, and Mobi for Kindle formats, ready for you to download at:

https://aka.ms/mspressfree

Check back often to see what is new!

Microsoft Virtual Academy

Build your knowledge of Microsoft technologies with free expert-led online training from Microsoft Virtual Academy (MVA). MVA offers a comprehensive library of videos, live events, and more to help you learn the latest technologies and prepare for certification exams. You'll find what you need here:

https://www.microsoftvirtualacademy.com

Quick access to online references

Throughout this book are addresses to webpages that the author has recommended you visit for more information. Some of these addresses (also known as URLs) can be painstaking to type into a web browser, so we've compiled all of them into a single list that readers of the print edition can refer to while they read.

Download the list at *https://aka.ms/examref762/downloads*.

The URLs are organized by chapter and heading. Every time you come across a URL in the book, find the hyperlink in the list to go directly to the webpage.

Errata, updates, & book support

We've made every effort to ensure the accuracy of this book and its companion content. You can access updates to this book—in the form of a list of submitted errata and their related corrections—at:

https://aka.ms/examref762/detail

If you discover an error that is not already listed, please submit it to us at the same page.

If you need additional support, email Microsoft Press Book Support at *mspinput@microsoft.com*.

Please note that product support for Microsoft software and hardware is not offered

through the previous addresses. For help with Microsoft software or hardware, go to *https://support.microsoft.com*.

We want to hear from you

At Microsoft Press, your satisfaction is our top priority, and your feedback our most valuable asset. Please tell us what you think of this book at:

https://aka.ms/tellpress

We know you're busy, so we've kept it short with just a few questions. Your answers go directly to the editors at Microsoft Press. (No personal information will be requested.) Thanks in advance for your input!

Stay in touch

Let's keep the conversation going! We're on Twitter: *http://twitter.com/MicrosoftPress*.

Important: How to use this book to study for the exam

Certification exams validate your on-the-job experience and product knowledge. To gauge your readiness to take an exam, use this Exam Ref to help you check your understanding of the skills tested by the exam. Determine the topics you know well and the areas in which you need more experience. To help you refresh your skills in specific areas, we have also provided "Need more review?" pointers, which direct you to more in-depth information outside the book.

The Exam Ref is not a substitute for hands-on experience. This book is not designed to teach you new skills.

We recommend that you round out your exam preparation by using a combination of available study materials and courses. Learn more about available classroom training at *https://www.microsoft.com/learning*. Microsoft Official Practice Tests are available for many exams at *https://aka.ms/practicetests*. You can also find free online courses and live events from Microsoft Virtual Academy at *https://www.microsoftvirtualacademy.com*.

This book is organized by the "Skills measured" list published for the exam. The "Skills measured" list for each exam is available on the Microsoft Learning website: *https://aka.ms/examlist*.

Note that this Exam Ref is based on this publicly available information and the author's experience. To safeguard the integrity of the exam, authors do not have access to the exam questions.

Chapter 1. Design and implement database objects

Developing and implementing a database for SQL Server starts with understanding both the process of designing a database and the basic structures that make up a database. A firm grip on those fundamentals is a must for an SQL Server developer, and is even more important for taking this exam.

Important Have you read page xv?

It contains valuable information regarding the skills you need to pass the exam.

We begin with the fundamentals of a typical database meant to store information about a business. This is generally referred to as online transaction processing (OLTP), where the goal is to store data that accurately reflects what happens in the business in a manner that works well for the applications. For this pattern, we review the relational database design pattern, which is covered in Skill 1.1. OLTP databases can be used to store more than business transactions, including the ability to store any data about your business, such as customer details, appointments, and so on.

Skills 1.2 and 1.3 cover some of the basic constructs, including indexes and views, that go into forming the physical database structures (Transact-SQL code) that applications use to create the foundational objects your applications use to do business.

In Skill 1.4 we explore columnstore indexes that focus strictly on analytics. While discussing analytics, we look at the de facto standard for building reporting structures called dimensional design. In dimensional design, the goal is to format the data in a form that makes it easier to extract results from large sets of data without touching a lot of different structures.

Skills in this chapter:

- Design and implement a relational database schema
- Design and implement indexes
- Design and implement views
- Implement columnstore indexes

Skill 1.1: Design and implement a relational database schema

In this section, we review some of the factors that go into creating the base tables that make up a relational database. The process of creating a relational database is not tremendously difficult. People build similar structures using Microsoft Excel every day. In this section, we are going to look at the basic steps that are needed to get started creating a database in

a professional manner.

This section covers how to:

- Design tables and schemas based on business requirements
- Improve the design of tables by using normalization
- Write create table statements
- Determine the most efficient data types to use

Designing tables and schemas based on business requirements

A very difficult part of any project is taking the time to gather business requirements. Not because it is particularly difficult in terms of technical skills, but because it takes lots of time and attention to detail. This exam that you are studying for is about developing the database, and the vast majority of topics center on the mechanical processes around the creation of objects to store and manipulate data via Transact-SQL code. However, the first few sections of this skill focus on required skills prior to actually writing Transact-SQL.

Most of the examples in this book, and likely on the exam, are abstract, contrived, and targeted to a single example; either using a sample database from Microsoft, or using examples that include only the minimal details for the particular concept being reviewed. There are, however, a few topics that require a more detailed narrative. To review the topic of designing a database, we need to start out with some basic requirements, using them to design a database that demonstrates database design concepts and normalization.

We have a scenario that defines a database need, including some very basic requirements. Questions on the exam can easily follow this pattern of giving you a small set of requirements and table structures that you need to match to the requirements. This scenario will be used as the basis for the first two sections of this chapter.

Imagine that you are trying to write a system to manage an inventory of computers and computer peripherals for a large organization. Someone has created a document similar in scope to the following scenario (realistic requirements are often hundreds or even thousands of pages long, but you can learn a lot from a single paragraph):

> *We have 1,000 computers, comprised of laptops, workstations, and tablets. Each computer has items associated with it, which we will list as mouse, keyboard, etc. Each computer has a tag number associated with it, and is tagged on each device with a tag graphic that can be read by tag readers manufactured by "Trey Research" (http://www.treyresearch.net/) or "Litware, Inc" (http://www.litwareinc.com/). Of course tag numbers are unique across tag readers. We don't know which employees are assigned which computers, but all computers that cost more than $300 are inventoried for the first three years after purchase using a different software system. Finally, employees need to*

have their names recorded, along with their employee number in this system.

Let's look for the tables and columns that match the needs of the requirements. We won't actually create any tables yet, because this is just the first step in the process of database design. In the next section, we spend time looking at specific tests that we apply to our design, followed by two sections on creating the table structures of a database.

The process of database design involves scanning requirements, looking for key types of words and phrases. For tables, you look for the nouns such as "computers" or "employee." These can be tables in your final database. Some of these nouns you discover in the requirements are simply subsets of one another: "computer" and "laptop." For example, laptop is not necessarily its own table at all, but instead may be just a type of computer. Whether or not you need a specific table for laptops, workstations, or tablets isn't likely to be important. The point is to match a possible solution with a set of requirements.

After scanning for nouns, you have your list of likely objects on which to save data. These will typically become tables after we complete our design, but still need to be refined by the normalization process that we will cover in the next section:

1. Computer
2. Employee

The next step is to look for attributes of each object. You do this by scanning the text looking for bits of information that might be stored about each object. For the Computer object, you see that there is a Type of Computer (laptop, workstation, or tablet), an Associated Item List, a Tag, a Tag Company, and a Tag Company URL, along with the Cost of the computer and employee that the computer is assigned to. Additionally, in the requirements, we also have the fact that they keep the computer inventoried for the first three years after purchase if it is > $300, so we need to record the Purchase Date. For the Employee object we are required to capture their Name and Employee Number.

Now we have the basic table structures to extract from the requirements, (though we still require some refinement in the following section on normalization) and we also define *schemas*, which are security/organizational groupings of tables and code for our implemented database. In our case, we define two schemas: Equipment and HumanResources.

Our design consists of the following possible tables and columns:

1. Equipment.Computer: (ComputerType, AssociatedItemList, Tag, TagCompany, TagCompanyURL, ComputerCost, PurchaseDate, AssignedEmployee)
2. HumanResources.Employee: (Name, EmployeeNumber)

The next step in the process is to look for how you would uniquely identify a row in your potential database. For example, how do you tell one computer from another. In the requirements, we are told that, "Each computer has a tag number," so we will identify that the Tag attribute must be unique for each Computer.

This process of designing the database requires you to work through the requirements until you have a set of tables that match the requirements you've been given.

In the real world, you don't alter the design from the provided requirements unless you discuss it with the customer. And in an exam question, you do whatever is written, regardless of whether it makes perfect sense. Do you need the URL of the TagCompany, for instance? If so, why? For the purposes of this exam, we will focus on the process of translating words into tables.

> **Note Logical Database Model**
>
> Our progress so far in designing this sample database is similar to what is referred to as a logical database model. For brevity, we have skipped some of the steps in a realistic design process. We continue to refine this example in upcoming sections.

Improving the design of tables by using normalization

Normalization is a set of "rules" that cover some of the most fundamental structural issues with relational database designs (there are other issues beyond normalization—for example, naming—that we do not talk about.) All of the rules are very simple at their core and each will deal with eliminating some issue that is problematic to the users of a database when trying to store data with the least redundancy and highest potential for performance using SQL Server 2016's relational engine.

The typical approach in database design is to work instinctively and then use the principles of normalization as a test to your design. You can expect questions on normalization to be similar, asking questions like, "is this a well-designed table to meet some requirement?" and any of the normal forms that might apply.

However, in this section, we review the normal forms individually, just to make the review process more straightforward. The rules are stated in terms of *forms*, some of which are numbered, and some which are named for the creators of the rule. The rules form a progression, with each rule becoming more and more strict. To be in a stricter normal form, you need to also conform to the lesser form, though none of these rules are ever followed one hundred percent of the time.

The most important thing to understand will be the concepts of normalization, and particularly how to verify that a design is normalized. In the following sections, we will review two families of normalization concepts:

- Rules covering the shape of a table
- Rules covering the relationship of non-key attributes to key attributes

Rules covering the shape of a table

A table's structure—based on what SQL Server (and most relational database management

systems, or RDBMSs) allow—is a very loose structure. Tables consist of rows and columns. You can put anything you want in the table, and you can have millions, even billions of rows. However, just because you *can* do something, doesn't mean it is correct.

The first part of these rules is defined by the mathematical definition of a relation (which is more or less synonymous with the proper structure of a table). Relations require that you have no duplicated rows. In database terminology, a column or set of columns that are used to uniquely identify one row from another is called a *key*. There are several types of keys we discuss in the following section, and they are all columns to identify a row (other than what is called a foreign key, which are columns in a table that reference another table's key attributes). Continuing with the example we started in the previous section, we have one such example in our design so far with: HumanResources.Employee: (Name, EmployeeNumber).

Using the Employee table definition that we started with back in the first section of this chapter, it would be allowable to have the following two rows of data represented:

Click here to view code image

```
Name                               EmployeeNumber
---------------------------------- --------------
Harmetz, Adam                      000010012
Harmetz, Adam                      000010012
```

This would not be a proper table, since you cannot tell one row from another. Many people try to fix this by adding some random bit of data (commonly called an *artificial key* value), like some auto generated number. This then provides a structure with data like the following, with some more data that is even more messed up, but still legal as the structure allows:

Click here to view code image

```
EmployeeId Name                               EmployeeNumber
---------- ---------------------------------- --------------------
----
         1 Harmetz, Adam                      000010012
         2 Harmetz, Adam                      000010012
         3 Popkova, Darya                     000000012
         4 Popkova, Darya                     000000013
```

In the next section on creating tables, we begin the review of ways we can enforce the uniqueness on data in column(s), but for now, let's keep it strictly in design mode. While this seems to make the table *better*, unless the EmployeeId column actually has some meaning to the user, all that has been done is to make the problem worse because someone looking for Adam's information can get one row or the other. What we really want is some sort of data in the table that makes the data unique based on data the user chooses. Name is not the correct choice, because two people can have the same name, but EmployeeNumber

is data that the user knows, and is used in an organization to identify an employee. A key like this is commonly known as a *natural key*. When your table is created, the artificial key is referred to as a *surrogate key*, which means it is a stand-in for the natural key for performance reasons. We talk more about these concepts in the "Determining the most efficient data types to use" section and again in Chapter 2, Skill 2.1 when choosing UNIQUE and PRIMARY KEY constraints.

After defining that EmployeeNumber must be unique, our table of data looks like the following:

Click here to view code image

```
EmployeeId Name                                EmployeeNumber
---------- ----------------------------------- ------------------
----
         1 Harmetz, Adam                       000010012
         2 Popkova, Darya                      000000013
```

The next two criteria concerning row shape are defined in the First Normal Form. It has two primary requirements that your table design must adhere to:

1. All columns must be atomic—that is, each column should represent one value
2. All rows of a table must contain the same number of values—no arrays

Starting with atomic column values, consider that we have a column in the Employee table we are working on that probably has a non-atomic value (*probably* because it is based on the requirements). Be sure to read the questions carefully to make sure you are not assuming things. The name column has values that contain a delimiter between what turns out to be the last name and first name of the person. If this is always the case then you need to record the first and last name of the person seperately. So in our table design, we will break 'Harmetz, Adam' into first name: 'Adam' and last name: 'Harmetz'. This is represented here:

Click here to view code image

```
EmployeeId LastName        FirstName          EmployeeNumber
---------- --------------- ------------------ ---------------
         1 Harmetz         Adam               000010012
         2 Popkova         Darya              000000013
```

For our design, let's leave off the EmployeeId column for clarity in the design. So the structure looks like:

Click here to view code image

```
HumanResources.Employee (EmployeeNumber [key], LastName,
FirstName)
```

Obviously the value here is that when you need to search for someone named 'Adam,'

you don't need to search on a partial value. Queries on partial values, particularly when the partial value does not include the leftmost character of a string, are not ideal for SQL Server's indexing strategies. So, the desire is that every column represents just a single value. In reality, names are always more complex than just last name and first name, because people have suffixes and titles that they really want to see beside their name (for example, if it was Dr. Darya Popkova, feelings could be hurt if the Dr. was dropped in correspondence with them.)

The second criteria for the first normal form is the rule about no repeating groups/arrays. A lot of times, the data that doesn't fit the atomic criteria is not different items, such as parts of a name, but rather it's a list of items that are the same types of things. For example, in our requirements, there is a column in the Computer table that is a list of items named AssociatedItemList and the example: 'mouse, keyboard.' Looking at this data, a row might look like the following:

Click here to view code image

```
Tag    AssociatedItemList
------ ------------------------------------
 s344  mouse, keyboard
```

From here, there are a few choices. If there are always two items associated to a computer, you might add a column for the first item, and again for a second item to the structure. But that is not what we are told in the requirements. They state: "Each computer has items associated with it." This can be any number of items. Since the goal is to make sure that column values are atomic, we definitely want to get rid of the column containing the delimited list. So the next inclination is to make a repeating group of column values, like:

Click here to view code image

```
Tag    AssociatedItem1 AssociatedItem2 ... AssociatedItemN
------ --------------- --------------- ... ----------------
 s344  mouse           keyboard        ... not applicable
```

This however, is not the desired outcome, because now you have created a fixed array of associated items with an index in the column name. It is very inflexible, and is limited to the number of columns you want to add. Even worse is that if you need to add something like a tag to the associated items, you end up with a structure that is very complex to work with:

Click here to view code image

```
Tag    AssociatedItem1 AssociatedItem1Tag AssociatedItem2
AssociatedItem2Tag
------ --------------- ------------------ --------------- -----
----------------
  s344 mouse           r232               keyboard        q472
```

Instead of this structure, create a new table that has a reference back to the original table, and the attributes that are desired:

```
Tag    AssociatedItem
------ -----------------
  s344 mouse
  s344 keyboard
```

So our object is: Equipment.ComputerAssociatedItem (Tag [Reference to Computer], AssociatedItem, [key Tag, AssociatedItem).

Now, if you need to search for computers that have keyboards associated, you don't need to either pick it out of a comma delimited list, nor do you need to look in multiple columns. Assuming you are reviewing for this exam, and already know a good deal about how indexes and queries work, you should see that everything we have done in this first section on normalization is going to be great for performance. The entire desire is to make scalar values that index well and can be searched for. It is never wrong to do a partial value search (if you can't remember how keyboard is spelled, for example, looking for associated items LIKE '%k%' isn't a violation of any moral laws, it just isn't a design goal that you are be trying to attain.

Rules covering the relationship of non-key attributes to key attributes

Once your data is shaped in a form that works best for the engine, you need to look at the relationship between attributes, looking for redundant data being stored that can get out of sync. In the first normalization section covering the shape of attributes, the tables were formed to ensure that each row in the structure was unique by choosing keys. For our two primary objects so far, we have:

Click here to view code image

```
HumanResources.Employee (EmployeeNumber)

Equipment.Computer (Tag)
```

In this section, we are going to look at how the other columns in the table relate to the key attributes. There are three normal forms that are related to this discussion:

- **Second Normal Form** All attributes must be a fact about the entire primary key and not a subset of the primary key.
- **Third Normal Form** All attributes must be a fact about the entire primary key, and not any non-primary key attributes

  For the second normal form to be a concern, you must have a table with multiple columns in the primary key. For example, say you have a table that defines a car parked in a parking space. This table can have the following columns:

  - **CarLicenseTag** (Key Column1)

- **SpaceNumber** (Key Column2)
- **ParkedTime**
- **CarColor**
- **CarModel**
- **CarManufacturer**
- **CarManufacturerHeadquarters**

Each of the nonkey attributes should say something about the combination of the two key attributes. The `ParkedTime` column is the time when the car was parked. This attribute makes sense. The others are all specifically about the car itself. So you need another table that looks like the following where all of the columns are moved to (the `CarLicenseTag` column stays as a reference to this new table. Now you have a table that represents the details about a car with the following columns:

- **CarLicenseTag** (Key Column)
- **CarColor**
- **CarModel**
- **CarManufacturer**
- **CarManufacturerHeadquarters**

Since there is a single key column, this must be in second normal form (like how the table we left behind with the `CarLicenseTag`, `SpaceNumber` and `ParkedTime` since `ParkedTime` references the entire key.) Now we turn our attention to the third normal form. Here we make sure that each attribute is solely focused on the primary key column. A car has a color, a model, and a manufacturer. But does it have a `CarManufacturerHeadquarters`? No, the manufacturer does. So you would create another table for that attribute and the key `CarManufacturer`. Progress through the design making more tables until you have eliminated redundancy.

The redundancy is troublesome because if you were to change the headquarter location for a manufacturer, you might need to do so for more than the one row or end up with mismatched data. Raymond Boyce and Edgar Codd (the original author of the normalization forms), refined these two normal forms into the following normal form, named after them:

- **Boyce-Codd Normal Form** Every candidate key is identified, all attributes are fully dependent on a key, and all columns must identify a fact about a key and nothing but a key.

All of these forms are stating that once you have set what columns uniquely define a row in a table, the rest of the columns should refer to what the key value represents. Continuing with the design based on the scenario/requirement we have used so far in the chapter,

consider the Equipment.Computer table. We have the following columns defined (Note that AssociatedItemList was removed from the table in the previous section):

Click here to view code image

```
Tag (key attribute), ComputerType, TagCompany, TagCompanyURL,
ComputerCost,
PurchaseDate, AssignedEmployee
```

In this list of columns for the Computer table, your job is to decide which of these columns describes what the Tag attribute is identifying, which is a computer. The Tag column value itself does not seem to describe the computer, and that's fine. It is a number that has been associated with a computer by the business in order to be able to tell two physical devices apart. However, for each of the other attributes, it's important to decide if the attribute describes something about the computer, or something else entirely. It is a good idea to take each column independently and think about what it means.

- **ComputerType** Describes the type of computer that is being inventoried.
- **TagCompany** The tag has a tag company, and since we defined that the tag number was unique across companies, this attribute is violating the Boyce-Codd Normal Form and must be moved to a different table.
- **TagCompanyURL** Much like TagCompany, the URL for the company is definitely not describing the computer.
- **ComputerCost** Describes how much the computer cost when purchased.
- **PurchaseDate** Indicates when the computer was purchased.
- **AssignedEmployee** This is a reference to the Employee structure. So while a computer doesn't really have an assigned employee in the real world, it does make sense in the overall design as it describes an attribute of the computer as it stands in the business.

Now, our design for these two tables looks like the following:

Click here to view code image

```
Equipment.Computer (Tag [key, ref to Tag], ComputerType,
ComputerCost, PurchaseDate,
AssignedEmployee [Reference to Employee]

Equipment.Tag (Tag [key], TagCompany, TagCompanyURL)
```

If the tables have the same key columns, do we need two tables? This depends on your requirements, but it is not out of the ordinary that you have two tables that are related to one another with a cardinality of one-to-one. In this case, you have a pool of tags that get created, and then assigned, to a device, or tags could have more than one use. Make sure to always take your time and understand the requirements that you are given with your question.

So we now have:

Click here to view code image

```
Equipment.Computer (Tag [key, Ref to Tag], ComputerType,
ComputerCost, PurchaseDate,
AssignedEmployee [Reference to Employee]
Equipment.TagCompany (TagCompany [key], TagCompanyURL)
Equipment.Tag (Tag [key], TagCompany [Reference to TagCompany])
```

And we have this, in addition to the objects we previously specified:

Click here to view code image

```
Equipment.ComputerAssociatedItem (Tag [Reference to Computer],
AssociatedItem, [key
Tag, AssociatedItem)

HumanResources.Employee (EmployeeNumber [key], LastName,
FirstName)
```

Generally speaking, the third normal form is referred to as the most important normal form, and for the exam it is important to understand that each table has one meaning, and each scalar attribute refers to the entire natural key of the final objects. Good practice can be had by working through tables in your own databases, or in our examples, such as the WideWorldImporters (the newest example database they have created), AdventureWorks, Northwind, or even Pubs. None of these databases are perfect, because doing an excellent job designing a database sometimes makes for really complex examples. Note that we don't have the detailed requirements for these sample databases. Don't be tricked by thinking you know what a system should look like by experience. The only thing worse than having no knowledge of your customer's business is having too much knowledge of their business.

Need More Review? Database Design and Normalization

What has been covered in this book is a very small patterns and techniques for database design that exist in the real world, and does not represent all of the normal forms that have been defined. Boyce-Codd/Third normal form is generally the limit of most writers. For more information on the complete process of database design, check out "Pro SQL Server Relational Database Design and Implementation," written by Louis Davidson for Apress in 2016. Or, for a more academic look at the process, get the latest edition of "An Introduction to Database Systems" by Chris Date with Pearson Press.

One last term needs to be defined: *denormalization*. After you have normalized your database, and have tested it out, there can be reasons to undo some of the things you have

done for performance. For example, later in the chapter, we add a formatted version of an employee's name. To do this, it duplicates the data in the LastName and FirstName columns of the table (in order to show a few concepts in implementation). A poor design for this is to have another column that the user can edit, because they might not get the name right. Better implementations are available in the implementation of a database.

Writing table create statements

The hard work in creating a database is done at this point of the process, and the process now is to simply translate a design into a physical database. In this section, we'll review the basic syntax of creating tables. In Chapter 2 we delve a bit deeper into the discussion about how to choose proper uniqueness constraints but we cover the mechanics of including such objects here.

Before we move onto CREATE TABLE statements, a brief discussion on object naming is useful. You sometimes see names like the following used to name a table that contain rows of purchase orders:

- PurchaseOrder
- PURCHASEORDER
- PO
- purchase_orders
- tbl_PurchaseOrder
- A12
- [Purchase Order] or "Purchase Order"

Of these naming styles, there are a few that are typically considered sub-optimal:

- **PO** Using abbreviations, unless universally acceptable tend to make a design more complex for newcomers and long-term users alike.
- **PURCHASEORDER** All capitals tends to make your design like it is 1970, which can hide some of your great work to make a modern computer system.
- **tbl_PurchaseOrder** Using a clunky prefix to say that this is a table reduces the documentation value of the name by making users ask what tbl means (admittedly this could show up in exam questions as it is not universally disliked).
- **A12** This indicates that this is a database where the designer is trying to hide the details of the database from the user.
- **[Purchase Order]** or **"Purchase Order"** Names that require delimiters, [brackets], or "double-quotes" are terribly hard to work with. Of the delimiter types, double-quotes are more standards-oriented, while the brackets are more typical SQL Server coding. Between the delimiters you can use any Unicode characters.

The more normal, programmer friendly naming standards are using Pascal-casing

(Leading character capitalized, words concatenated: PurchaseOrder), Camel Casing (leading character lower case: purchaseOrder), or using underscores as delimiters (purchase_order).

> **Need More Review? Database Naming Rules**
>
> This is a very brief review of naming objects. Object names must fall in the guidelines of a database identifier, which has a few additional rules. You can read more about database identifiers here in this MSDN article: *https://msdn.microsoft.com/en-us/library/ms175874.aspx*.

Sometimes names are plural, and sometimes singular, and consistency is the general key. For the exam, there are likely to be names of any format, plural, singular, or both. Other than interpreting the meaning of the name, naming is not listed as a skill.

To start with, create a schema to put objects in. Schemas allow you to group together objects for security and logical ordering. By default, there is a schema in every database called dbo, which is there for the database owner. For most example code in this chapter, we use a schema named Examples located in a database named ExamBook762Ch1, which you see referenced in some error messages.

Click here to view code image

```
CREATE SCHEMA Examples;
GO --CREATE SCHEMA must be the only statement in the batch
```

The CREATE SCHEMA statement is terminated with a semicolon at the end of the statement. All statements in Transact-SQL can be terminated with a semicolon. While not all statements must end with a semicolon in SQL Server 2016, not terminating statements with a semicolon is a deprecated feature, so it is a good habit to get into. GO is not a statement in Transact-SQL it is a batch separator that splits your queries into multiple server communications, so it does not need (or allow) termination.

To create our first table, start with a simple structure that's defined to hold the name of a widget, with attributes for name and a code:

Click here to view code image

```
CREATE TABLE Examples.Widget
(
    WidgetCode  varchar(10) NOT NULL
            CONSTRAINT PKWidget PRIMARY KEY,
    WidgetName  varchar(100) NULL
);
```

Let's break down this statement into parts:

Click here to view code image

```
CREATE TABLE Examples.Widget
```

Here we are naming the table to be created. The name of the table must be unique from all other object names, including tables, views, constraints, procedures, etc. Note that it is a best practice to reference all objects explicitly by at least their two-part names, which includes the name of the object prefixed with a schema name, so most of the code in this book will use two-part names. In addition, object names that a user may reference directly such as tables, views, stored procedures, etc. have a total of four possible parts. For example, Server.Database.Schema.Object has the following parts:

- **Server** The local server, or a linked server name that has been configured. By default, the local server from which you are executing the query.
- **Database** The database where the object you are addressing resides. By default, this is the database that to which you have set your context.
- **Schema** The name of the schema where the object you are accessing resides within the database. Every login has a default schema which defaults to dbo. If the schema is not specified, the default schema will be searched for a matching name.
- **Object** The name of the object you are accessing, which is not optional.

In the CREATE TABLE statement, if you omit the schema, it is created in the default schema. So the CREATE TABLE Widget would, by default, create the table dbo.Widget in the database of context. You can create the table in a different database by specifying the database name: CREATE TABLE Tempdb..Widget or Tempdb.dbo.Widget. There is an article here: (*https://technet.microsoft.com/en-us/library/ms187879.aspx*.) from an older version of books online that show you the many different forms of addressing an object.

The next line:

Click here to view code image

```
    WidgetCode  varchar(10) NOT NULL
```

This specifies the name of the column, then the data type of that column. There are many different data types, and we examine their use and how to make the best choice in the next section. For now, just leave it as this determines the format of the data that is stored in this column. NOT NULL indicates that you must have a known value for the column. If it simply said NULL, then it indicates the value of the column is allowed to be NULL.

NULL is a special value that mathematically means UKNOWN. A few simple equations that can help clarify NULL is that: UNKNOWN + any value = UNKNOWN, and NOT(UNKNOWN) = UNKNOWN. If you don't know a value, adding any other value to it is still unknown. And if you don't know if a value is TRUE or FALSE, the opposite of that is still not known. In comparisons, A NULL expression is never equivalent to a NULL expression. So if you have the following conditional: IF (NULL = NULL); the expression would not be TRUE, so it would not succeed.

If you leave off the NULL specification, whether or not the column allows NULL values is based on a couple of things. If the column is part of a PRIMARY KEY constraint that is being added in the CREATE TABLE statement (like in the next line of code), or the setting: SET ANSI_NULL_DFLT_ON, then NULL values are allowed.

Note NULL Specification

For details on the SET ANSI_NULL_DFLT_ON setting, go to *https://msdn.microsoft.com/en-us/library/ms187375.aspx*.). It is considered a best practice to always specify a NULL specification for columns in your CREATE and ALTER table statements.

The following line of code is a continuation of the previous line of code, since it was not terminated with a comma (broken out to make it easier to explain):

Click here to view code image

```
                    CONSTRAINT PKWidget PRIMARY KEY,
```

This is how you add a constraint to a single column. In this case, we are defining that the WidgetCode column is the only column that makes up the primary key of the table. The CONSTRAINT PKWidget names the constraint. The constraint name must be unique within the schema, just like the table name. If you leave the name off and just code it as PRIMARY KEY, SQL Server provides a name that is guaranteed unique, such as PK__Widget__1E5F7A7F7A139099. Such a name changes every time you create the constraint, so it's really only suited to temporary tables (named either with # or ## as a prefix for local or global temporary objects, respectively).

Alternatively, this PRIMARY KEY constraint could have been defined independently of the column definition as (with the leading comma there for emphasis):

Click here to view code image

```
                    ,CONSTRAINT PKWidget PRIMARY KEY
(WidgetCode),
```

This form is needed when you have more than one column in the PRIMARY KEY constraint, like if both the WidgetCode and WidgetName made up the primary key value:

Click here to view code image

```
                    ,CONSTRAINT PKWidget PRIMARY KEY
(WidgetCode, WidgetName),
```

This covers the simple version of the CREATE TABLE statement, but there are a few additional settings to be aware of. First, if you want to put your table on a file group other than the default one, you use the ON clause:

Click here to view code image

```
CREATE TABLE Examples.Widget
(
    WidgetCode  varchar(10) NOT NULL
           CONSTRAINT PKWidget PRIMARY KEY,
    WidgetName  varchar(100) NULL
) ON FileGroupName;
```

There are also table options for using temporal extensions, as well as partitioning. These are not a part of this exam, so we do not cover them in any detail, other than to note their existence.

In addition to being able to use the CREATE TABLE statement to create a table, it is not uncommon to encounter the ALTER TABLE statement on the exam to add or remove a constraint. The ALTER TABLE statement allows you to add columns to a table and make changes to some settings.

For example, you can add a column using:

Click here to view code image

```
ALTER TABLE Examples.Widget
    ADD NullableColumn int NULL;
```

If there is data in the table, you either have to create the column to allow NULL values, or create a DEFAULT constraint along with the column (which is covered in greater detail in Chapter 2, Skill 2.1).

Click here to view code image

```
ALTER TABLE Examples.Widget
    ADD NotNullableColumn int NOT NULL
        CONSTRAINT DFLTWidget_NotNullableColumn DEFAULT ('Some
Value');
To drop the column, you need to drop referencing constraints,
which you also do with the
ALTER TABLE statement:

ALTER TABLE Examples.Widget
    DROP DFLTWidget_NotNullableColumn;
```

Finally, we will drop this column (because it would be against the normalization rules we have discussed to have this duplicated data) using:

Click here to view code image

```
ALTER TABLE Examples.Widget
    DROP COLUMN NotNullableColumn;
```

Need More Review? Creating and Altering Tables

We don't touch on everything about the CREATE TABLE or ALTER TABLE

statement, but you can read more about the various additional settings you can see in Books Online in the CREATE TABLE (*https://msdn.microsoft.com/en-us/library/ms174979.aspx*) and ALTER TABLE (*https://msdn.microsoft.com/en-us/library/ms190273.aspx*) topics.

Determining the most efficient data types to use

Every column in a database has a data type, which is the first in a series of choices to limit what data can be stored. There are data types for storing numbers, characters, dates, times, etc., and it's your job to make sure you have picked the very best data type for the need. Choosing the best type has immense value for the systems implemented using the database.

- **It serves as the first limitation of domain of data values that the columns can store.** If the range of data desired is the name of the days of the week, having a column that allows only integers is completely useless. If you need the values in a column to be between 0 and 350, a tinyint won't work because it has a maximum of 256, so a better choice is smallint, that goes between –32,768 and 32,767, In Chapter 2, we look at several techniques using CONSTRAINT and TRIGGER objects to limit a column's value even further.
- **It is important for performance** Take a value that represents the 12th of July, 1999. You could store it in a char(30) as '12th of July, 1999', or in a char(8) as '19990712'. Searching for one value in either case requires knowledge of the format, and doing ranges of date values is complex, and even very costly, performance-wise. Using a date data type makes the coding natural for the developer and the query processor.

When handled improperly, data types are frequently a source of interesting issues for users. Don't limit data enough, and you end up with incorrect, wildly formatted data. Limit too much, like only allowing 35 letters for a last name, and Janice "Lokelani" Keihanaikukauakahihuliheekahaunaele has to have her name truncated on her driver's license (true story, as you can see in the following article on USA Today *http://www.usatoday.com/story/news/nation/2013/12/30/hawaii-long-name/4256063/*).

SQL Server has an extensive set of data types that you can choose from to match almost any need. The following list contains the data types along with notes about storage and purpose where needed.

- **Precise Numeric** Stores number-based data with loss of precision in how it stored.
  - **bit** Has a domain of 1, 0, or NULL; Usually used as a pseudo-Boolean by using 1 = True, 0 = False, NULL = Unknown. Note that some typical integer operations, like basic math, cannot be performed. (1 byte for up to 8 values)
  - **tinyint** Integers between 0 and 255 (1 byte).
  - **smallint** Integers between –32,768 and 32,767 (2 bytes).

- **int** Integers between 2,147,483,648 to 2,147,483,647 (–2^31 to 2^31 – 1) (4 bytes).
- **bigint** Integers between 9,223,372,036,854,775,808 to 9,223,372,036,854,775,807 (-2^63 to 2^63 – 1) (8 bytes).
- **decimal (or numeric which are functionally the same, with decimal the more standard type)**: All numbers between –10^38 – 1 and 10^38 – 1, with a fixed set of digits up to 38. decimal(3,2) would be a number between -9.99 and 9.99. And decimal(38,37), with be a number with one digit before the decimal point, and 37 places after it. Uses between 5 and 17 bytes, depending on precision.
- **money** Monetary values from –922,337,203,685,477.5808 through 922,337,203,685,477.5807 (8 bytes).
- **smallmoney** Money values from –214,748.3648 through 214,748.3647 (4 bytes).

- **Approximate numeric data** Stores approximations of numbers based on IEEE 754 standard, typically for scientific usage. Allows a large range of values with a high amount of precision but you lose precision of very large or very small numbers.
  - **float(N)** Values in the range from –1.79E + 308 through 1.79E + 308 (storage varies from 4 bytes for N between 1 and 24, and 8 bytes for N between 25 and 53).
  - **real** Values in the range from –3.40E + 38 through 3.40E + 38. real is an ISO synonym for a float(24) data type, and hence equivalent (4 bytes).
- **Date and time values** Stores values that deal storing a point in time.
  - **date** Date-only values from January 1, 0001, to December 31, 9999 (3 bytes).
  - **time(N)** Time-of-day-only values with N representing the fractional parts of a second that can be stored. time(7) is down to HH:MM:SS.0000001 (3 to 5 bytes).
  - **datetime2(N)** This type stores a point in time from January 1, 0001, to December 31, 9999, with accuracy just like the *time* type for seconds (6 to 8 bytes).
  - **datetimeoffset** Same as datetime2, plus includes an offset for time zone offset (does not deal with daylight saving time) (8 to 10 bytes).
  - **smalldatetime** A point in time from January 1, 1900, through June 6, 2079, with accuracy to 1 minute (4 bytes).
  - **datetime** Points in time from January 1, 1753, to December 31, 9999, with accuracy to 3.33 milliseconds (so the series of fractional seconds starts as: .003, .007, .010, .013, .017 and so on) (8 bytes).
- **Binary data** Strings of bits used for storing things like files, encrypted values, etc. Storage for these data types is based on the size of the data stored in bytes, plus any overhead for variable length data.
  - **binary(N)** Fixed-length binary data with a maximum value of N of 8000, for an 8,000 byte long binary value.

- **varbinary(N)** Variable-length binary data with maximum value of N of 8,000.
- **varbinary(max)** Variable-length binary data up to (2^31) – 1 bytes (2GB) long. Values are often stored using filestream filegroups, which allow you to access files directly via the Windows API, and directly from the Windows File Explorer using filetables.

- **Character (or string) data** String values, used to store text values. Storage is specified in number of characters in the string.
  - **char(N)** Fixed-length character data up to 8,000 characters long. When using fixed length data types, it is best if most of the values in the column are the same, or at least use most of the column.
  - **varchar(N)** Variable-length character data up to 8,000 characters long.
  - **varchar(max)** Variable-length character data up to (2^31) – 1 bytes (2GB) long. This is a very long string of characters, and should be used with caution as returning rows with 2GB per row can be hard on your network connection.
  - **nchar, nvarchar, nvarchar(max)** Unicode equivalents of char, varchar, and varchar(max). Unicode is a double (and in some cases triple) byte character set that allows for more than the 256 characters at a time that the ASCII characters do. Support for Unicode is covered in detail in this article: *https://msdn.microsoft.com/en-us/library/ms143726.aspx*. It is generally accepted that it is best to use Unicode when storing any data where you have no control over the data that is entered. For example, object names in SQL Server allow Unicode names, to support most any characters that a person might want to use for names. It is very common that columns for people's names are stored in Unicode to allow for a full range of characters to be stored.
- **Other data types** Here are a few more data types:
  - **sql_variant** Stores nearly any data type, other than CLR based ones like hierarchyId, spatial types, and types with a maximum length of over 8016 bytes. Infrequently used for patterns where the data type of a value is unknown before design time.
  - **rowversion** (**timestamp** is a synonym) Used for optimistic locking to version-stamp in a row. The value in the rowversion data type-based column changes on every modification of the row. The name of this type was timestamp in all SQL Server versions before 2000, but in the ANSI SQL standards, the timestamp type is equivalent to the datetime data type. Stored as a 16-byte binary value.
  - **uniqueidentifier** Stores a globally unique identifier (GUID) value. A GUID is a commonly used data type for an artificial key, because a GUID can be generated by many different clients and be almost 100 percent assuredly unique. It has downsides of being somewhat random when being sorted in generated order, which

can make it more difficult to index. We discuss indexing in Skill 1.2. Represented as a 36-character string, but is stored as a 16-byte binary value.

- **XML** Allows you to store an XML document in a column value. The XML type gives you a rich set of functionality when dealing with structured data that cannot be easily managed using typical relational tables.
- **Spatial types** (geometry, geography, circularString, compoundCurve, and curvePolygon) Used for storing spatial data, like for shapes, maps, lines, etc.
- **heirarchyId** Used to store data about a hierarchy, along with providing methods for manipulating the hierarchy.

Need More Review Data type Overview

This is just an overview of the data types. For more reading on the types in the SQL Server Language Reference, visit the following URL: *https://msdn.microsoft.com/en-us/library/ms187752.aspx*.

The difficultly in choosing the data type is that you often need to consider not just the requirements given, but real life needs. For example, say we had a table that represents a company and all we had was the company name. You might logically think that the following makes sense:

Click here to view code image

```
CREATE TABLE Examples.Company
(
         CompanyName    varchar(50) NOT NULL
                        CONSTRAINT PKCompany PRIMARY KEY
);
```

There are a few concerns with this choice of data type. First let's consider the length of a company name. Almost every company name will be shorter than 50 characters. But there are definitely companies that exist with much larger names than this, even if they are rare. In choosing data types, it is important to understand that you have to design your objects to allow the maximum size of data *possible*. If you could ever come across a company name that is greater than 50 characters and need to store it completely, this will not do. The second concern is character set. Using ASCII characters is great when all characters will be from A-Z (upper or lower case), and numbers. As you use more special characters, it becomes very difficult because there are only 256 ASCII characters per code page.

In an exam question, if the question was along the lines of "the 99.9 percent of the data that goes into the CompanyName column is 20 ASCII characters or less, but there is one row that has 2000 characters with Russian and Japanese characters, what data type would you use?" the answer would be nvarchar(2000). varchar(2000) would not have the right character set, nchar(2000) would be wasteful, and integer would be just plain silly.

Note Column Details

For the exam, expect more questions along the lines of whether a column should be one version of a type or another, like varchar or nvarchar. Most any column where you are not completely in control of the values for the data (like a person's name, or external company names) should use Unicode to give the most flexibility regarding what data can go into the column.

There are several groups of data types to learn in order to achieve a deep understanding. For example, consider a column named Amount in a table of payments that holds the amount of a payment:

Click here to view code image

```
CREATE TABLE Examples.Payment
(
          PaymentNumber char(10) NOT NULL
                        CONSTRAINT PKPayment PRIMARY KEY,
          Amount int NOT NULL
);
```

Does the integer hold an amount? Definitely. But in most countries, monetary units are stored with a fractional part, and while you could shift the decimal point in the client, that is not the best design. What about a real data type? Real types are meant for scientific amounts where you have an extremely wide amount of values that could meet your needs, not for money where fractional parts, or even more, could be lost in precision. Would decimal(30,20) be better? Clearly. But it isn't likely that most organizations are dealing with 20 decimal places for monetary values. There is also a money data type that has 4 decimal places, and something like decimal(10,2) also works for most monetary cases. Actually, it works for any decimal or numeric types with a scale of 2 (in decimal(10,2), the 10 is the precision or number of digits in the number; and 2 is the scale, or number of places after the decimal point).

The biggest difficulty with choosing a data type goes back to the requirements. If there are given requirements that say to store a company name in 10 characters, you use 10 characters. The obvious realization is that a string like 'Blue Yonder Airlines' takes more than 10 characters (even if it is fictitious, you know real company names that won't fit in 10 characters). You should default to what the requirements state (and in the non-exam world verify it with the customer.) All of the topics in this Skill 1.1 section, and on the exam should be taken from the requirements/question text. If the client gives you specific specifications to follow, you follow them. If the client says "store a company name" and gives you no specific limits, then you use the best data type. The exam is multiple choice, so unlike a job interview where you might be asked to give your reasoning, you just choose a *best* answer.

In Chapter 2, the first of the skills covered largely focuses on refining the choices in this section. For example, say the specification was to store a whole number between -20 and 2,000,000,000. The int data type stores all of those values, but also stores far more value. The goal is to make sure that 100 percent of the values that are stored meet the required range. Often we need to limit a value to a set of values in the same or a different table. Data type alone doesn't do it, but it gets you started on the right path, something you could be asked.

Beyond the basic data type, there are a couple of additional constructs that extend the concept of a data type. They are:

- **Computed Columns** These are columns that are based on an expression. This allows you to use any columns in the table to form a new value that combines/reformats one or more columns.
- **Dynamic Data Masking** Allows you to mask the data in a column from users, allowing data to be stored that is private in ways that can show a user parts of the data.

Computed columns

Computed columns let you manifest an expression as a column for usage (particularly so that the engine maintains values for you that do not meet the normalization rules we discussed earlier). For example, say you have a table with columns FirstName and LastName, and want to include a column named FullName. If FullName was a column, it would be duplicated data that we would need to manage and maintain, and the values could get out of sync. But adding it as a computed column means that the data is either be instantiated at query time or, if you specify it and the expression is deterministic, persisted. (A deterministic calculation is one that returns the same value for every execution. For example, the COALESCE() function, which returns the first non-NULL value in the parameter list, is deterministic, but the GETDATE() function is not, as every time you perform it, you could get a different value.)

So we can create the following:

Click here to view code image

```
CREATE TABLE Examples.ComputedColumn
(
     FirstName  nvarchar(50) NULL,
     LastName   nvarchar(50) NOT NULL,
     FullName AS CONCAT(LastName,',' + FirstName)
);
```

Now, in the FullName column, we see either the LastName or LastName, FirstName for each person in our table. If you added PERSISTED to the end of the declaration, as in:

Click here to view code image

```
ALTER TABLE Examples.ComputedColumn DROP COLUMN FullName;

ALTER TABLE Examples.ComputedColumn
   ADD FullName AS CONCAT(LastName,', ' + FirstName) PERSISTED;
```

Now the expression be evaluated during access in a statement, but is saved in the physical table storage structure along with the rest of the data. It is read only to the programmer's touch, and it's maintained by the engine. Throughout this book, one of the most important tasks for you as an exam taker is to be able to predict the output of a query, based on structures and code. Hence, when we create an object, we provide a small example explaining it. This does not replace having actually attempted everything in the book on your own (many of which you will have done professionally, but certainly not all.) These examples should give you reproducible examples to start from. In this case, consider you insert the following two rows:

Click here to view code image

```
INSERT INTO Examples.ComputedColumn
VALUES (NULL,'Harris'),('Waleed','Heloo');
```

Then query the data to see what it looks like with the following SELECT statement.

Click here to view code image

```
SELECT *
FROM   Examples.ComputedColumn;
```

You should be able to determine that the output of the statement has one name for Harris, but two comma delimited names for Waleed Heloo.

Click here to view code image

```
FirstName    LastName       FullName
------------ -------------- --------------------
NULL         Harris         Harris
Waleed       Heloo          Heloo, Waleed
```

Dynamic data masking

Dynamic data masking lets you mask data in a column from the view of the user. So while the user may have all rights to a column, (INSERT, UPDATE, DELETE, SELECT), when they use the column in a SELECT statement, instead of showing them the actual data, it masks it from their view. For example, if you have a table that has email addresses, you might want to mask the data so most users can't see the actual data when they are querying the data. In Books Online, the topic of Dynamic Data Masking falls under security (*https://msdn.microsoft.com/en-us/library/mt130841.aspx*), but as we will see, it doesn't behave like classic security features, as you will be adding some code to the DDL of the table, and there isn't much fine tuning of the who can access the unmasked value.

As an example, consider the following table structure, with three rows to use to show the feature in action:

Click here to view code image

```
CREATE TABLE Examples.DataMasking
(
    FirstName    nvarchar(50) NULL,
    LastName    nvarchar(50) NOT NULL,
    PersonNumber char(10) NOT NULL,
    Status    varchar(10), --domain of values
('Active','Inactive','New')
    EmailAddress nvarchar(50) NULL, --(real email address ought
to be longer)
    BirthDate date NOT NULL, --Time we first saw this person.
    CarCount   tinyint NOT NULL --just a count we can mask
);

INSERT INTO
Examples.DataMasking(FirstName,LastName,PersonNumber, Status,
                                    EmailAddress, BirthDate,
CarCount)
VALUES('Jay','Hamlin','0000000014','Active','jay@litwareinc.com
01-12',0),
    ('Darya','Popkova','0000000032','Active','darya.p@proseware
05-22', 1),
    ('Tomasz','Bochenek','0000000102','Active',NULL, '1959-03-
30', 1);
```

There are four types of data mask functions that we can apply:

- **Default** Takes the default mask of the data type (not of the DEFAULT constraint of the column, but the data type).
- **Email** Masks the email so you only see a few meaningful characters.
- **Random** Masks any of the numeric data types (int, smallint, decimal, etc) with a random value within a range.
- **Partial** Allows you to take values from the front and back of a value, replacing the center with a fixed string value.

Once applied, the masking function emits a masked value unless the column value is NULL, in which case the output is NULL.

Who can see the data masked or unmasked is controlled by a database level permission called UNMASK. The dbo user always has this right, so to test this, we create a different user to use after applying the masking. The user must have rights to SELECT data from the table:

Click here to view code image

```
CREATE USER MaskedView WITHOUT LOGIN;
GRANT SELECT ON Examples.DataMasking TO MaskedView;
```

The first masking type we apply is default. This masks the data with the default for the particular data type (not the default of the column itself from any DEFAULT constraint if one exists). It is applied using the ALTER TABLE...ALTER COLUMN statement, using the following syntax:

Click here to view code image

```
ALTER TABLE Examples.DataMasking ALTER COLUMN FirstName
    ADD MASKED WITH (FUNCTION = 'default()');
ALTER TABLE Examples.DataMasking ALTER COLUMN BirthDate
    ADD MASKED WITH (FUNCTION = 'default()');
```

Now, when someone without the UNMASK database right views this data, it will make the FirstName column value look like the default for string types which is 'XXXX', and the date value will appear to all be '1900-01-01'. Note that care should be taken that when you use a default that the default value isn't used for calculations. Otherwise you could send a birthday card to every customer on Jan 1, congratulating them on being over 116 years old.

> **Note The MASKED WITH Clause**
>
> To add masking to a column in the CREATE TABLE statement, the MASKED WITH clause goes between the data type and NULL specification. For example: LastName nvarchar(50) MASKED WITH (FUNCTION = 'default()') NOT NULL

Next, we add masking to the EmailAddress column. The email filter has no configuration, just like default(). The email() function uses fixed formatting to show the first letter of an email address, always ending in the extension .com:

Click here to view code image

```
ALTER TABLE Examples.DataMasking ALTER COLUMN EmailAddress
    ADD MASKED WITH (FUNCTION = 'email()');
```

Now the email address: darya.p@proseware.net will appear as dXXX@XXXX.com. If you wanted to mask the email address in a different manner, you could also use the following masking function.

The partial() function is by far the most powerful. It let's you take the number of characters from the front and the back of the string. For example, in the following data mask, we make the PersonNumber show the first and last characters. This column is of a fixed width, so the values will show up as the same size as previously.

Click here to view code image

```
--Note that it uses double quotes in the function call
ALTER TABLE Examples.DataMasking ALTER COLUMN PersonNumber
    ADD MASKED WITH (FUNCTION = 'partial(2,"*******",1)');
```

The size of the mask is up to you. If you put fourteen asterisks, the value would look fourteen wide. Now, PersonNumber: '0000000102' looks like '00*******2', as does: '0000000032'. Apply the same sort of mask to a non-fixed length column, the output will be fixed width if there is enough data for it to be:

Click here to view code image

```
ALTER TABLE Examples.DataMasking ALTER COLUMN LastName
    ADD MASKED WITH (FUNCTION = 'partial(3,"_____",2)');
```

Now 'Hamlin' shows up as 'Ham_____n'. Partial can be used to default the entire value as well, as if you want to make a value appear as unknown. The partial function can be used to default the entire value as well. In our example, you default the Status value to 'Unknown':

Click here to view code image

```
ALTER TABLE Examples.DataMasking ALTER COLUMN Status
    ADD MASKED WITH (Function = 'partial(0,"Unknown",0)');
```

Finally, to the CarCount column, we will add the random() masking function. It will put a random number of the data type of the column between the start and end value parameters:

Click here to view code image

```
ALTER TABLE Examples.DataMasking ALTER COLUMN CarCount
    ADD MASKED WITH (FUNCTION = 'random(1,3)');
```

Viewing the data as dbo (which you typically will have when designing and building a database):

Click here to view code image

```
SELECT *
FROM   Examples.DataMasking;
```

There is no apparent change:

Click here to view code image

```
FirstName LastName  PersonNumber
Status     EmailAddress           BirthDate  CarCount
--------- --------- ----------- ---------- ------------------
--- ---------- --------
Jay       Hamlin    0000000014   Active     jay@litwareinc.com
01-12 0
```

```
Darya     Popkova   0000000032   Active     darya.p@proseware.n
05-22 1
Tomasz    Bochenek  0000000102   Active     NULL
03-30 1
```

Now, using the EXECUTE AS statement to impersonate this MaskedView user, run the following statement:

Click here to view code image

```
EXECUTE AS USER = 'MaskedView';
SELECT *
FROM   Examples.DataMasking;

FirstName LastName     PersonNumber
Status  EmailAddress              BirthDate  CarCount
--------- ------------ ------------ ------- ------------------
----- ---------- --------
xxxx      Hamlin       00****14     Unknown
jXXX@XXXX.com            1900-01-01 2
xxxx      Popkova      00****32     Unknown
dXXX@XXXX.com            1900-01-01 1
xxxx      Bochenek     00****02     Unknown
NULL                     1900-01-01 1
```

Run the statement multiple times and you will see the CarCount value changing multiple times. Use the REVERT statement to go back to your normal user context, and check the output of USER_NAME() to make sure you are in the correct context, which should be dbo for these examples:

```
REVERT; SELECT USER_NAME();
```

Skill 1.2: Design and implement indexes

In this section, we examine SQL Server's B-Tree indexes on on-disk tables. In SQL Server 2016, we have two additional indexing topics, covered later in the book, those being columnstore indexes (Skill 1.4) and indexes on memory optimized tables (Skill 3.4). A term that will be used for the B-Tree based indexes is *rowstore*, in that their structures are designed to contain related data for a row together. Indexes are used to speed access to rows using a *scan* of the values in a table or index, or a *seek* for specific row(s) in an index.

Indexing is a very complex topic, and a decent understanding of the internal structures makes understanding when to and when not to use an index easier. Rowstore indexes on the on-disk tables are based on the concept of a B-Tree structure, consisting of index nodes that sort the data to speed finding one value. Figure 1-1 shows the basic structure of all of these types of indexes.

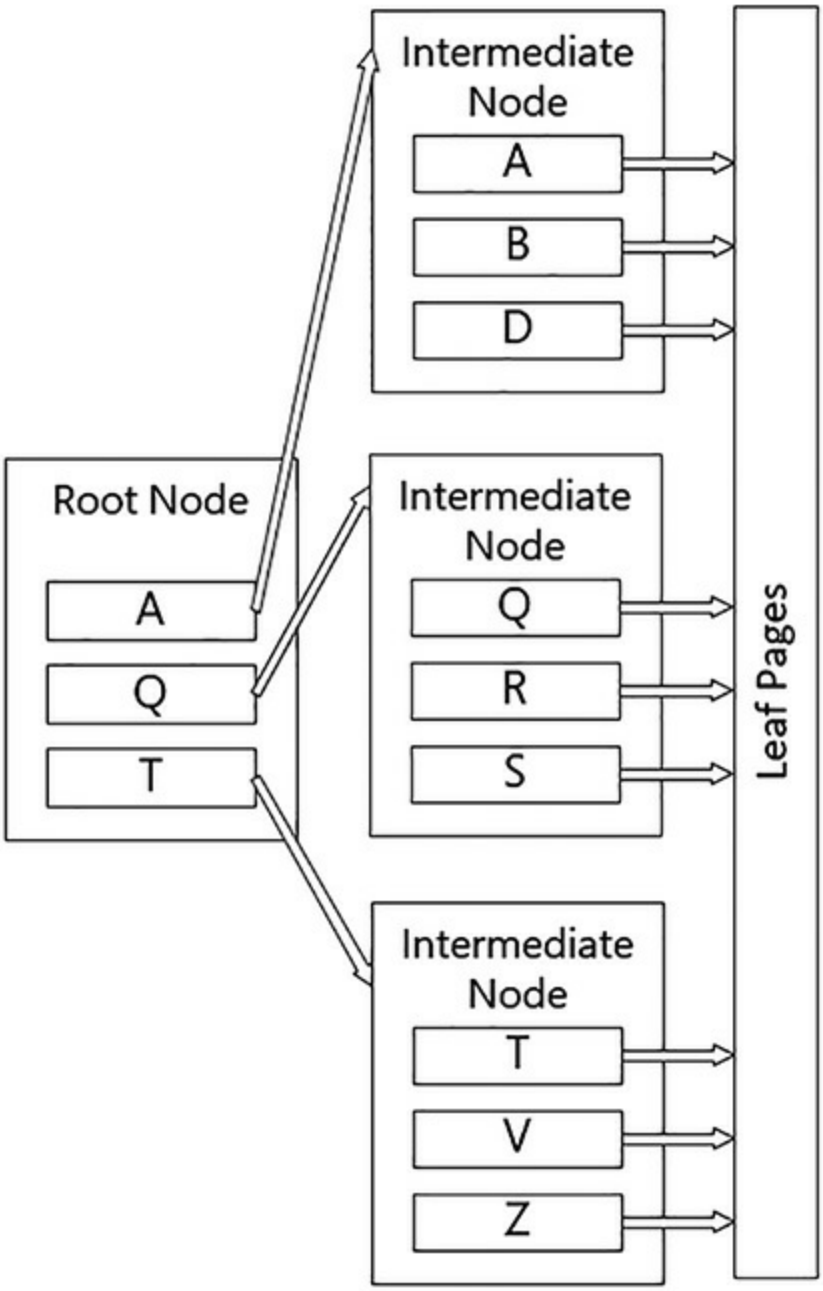


FIGURE 1-1 The base structure of a B-Tree Index

In the index shown in Figure 1-1, when you search for an item, if it is between A and Q, you follow the pointer to the first intermediate node of the tree. This structure is repeated for as many levels as there are in the index. When you reach the last intermediate node (which may be the root node for smaller indexes), you go to the leaf node.

There are two types of indexes in SQL Server: *clustered* and *non-clustered*. Clustered indexes are indexes where the leaf node in the tree contains the actual data in the table (A table without a clustered index is a *heap* which is made up of non-sequential, 8K pages of data.) A non-clustered index is a separate structure that has a copy of data in the leaf node that is in the keys, along with a pointer to the heap or clustered index.

The structure of the non-clustered leaf pages depends on whether the table is a heap or a

clustered table. For a heap, it contains a pointer to the physical structure where the data resides. For a clustered table, it contains the value of the clustered index keys (referred to as the *clustering key*.) Last, for a clustered columnstore index, it is the position in the columnstore index (covered in Skill 1.4).

When the index key is a single column, it is referred to as a *simple index*, and when there are multiple columns, it is called a *composite index*. The index nodes (and leaf pages) will be sorted in the order of the leading column first, then the second column, etc. For a composite index it is best to choose the column that is the most *selective* for the lead column, which is to say, it has the most unique values amongst the rows of the table.

The limit on the size of index key (for the data in all of the columns declared for the index) is based on the type of the index. The maximum key size for a non-clustered index is 1700 bytes, and 900 for a clustered index. Note that the smaller the index key size, the more that fits on each index level, and the fewer index levels, the fewer reads per operation. A page contains a maximum of 8060 bytes, and there is some overhead when storing variable length column values. If your index key values are 1700 bytes, which means you could only have 4 rows per page. In a million row table, you can imagine this would become quite a large index structure.

Need More Review? Indexing

For more details on indexes that we will use in this skill, and some that we will cover later in the book, MSDN has a set of articles on indexes lined to from this page: *https://msdn.microsoft.com/en-us/library/ms175049.aspx*.

This section covers how to:

- Design new indexes based on provided tables, queries, or plans
- Distinguish between indexed columns and included columns
- Implement clustered index columns by using best practices
- Recommend new indexes based on query plans

Design new indexes based on provided tables, queries, or plans

There are two phases of a project where you typically add indexes during the implementation process:

- During the database design phase
- During the coding phase, continuing throughout the lifecycle of your implementation

The primary difference between the two phases is need. During the design phase, there are constraints that create indexes as part of their creation, and a few situations where it is essential to create an index without even executing a query. After you have configured your

tables, the goal of indexes is almost completely aligned to how well your queries work, and you must add indexes where the need arises, and not just because it seems like a good idea.

> **Note Concurrency Concepts**
>
> Chapter 3 reviews the concepts of concurrency, and the first step to building highly concurrent database systems is to get the design right, and match the indexing of the database to the user's queries, so queries only access the minimum amount of data needed to answer queries.

Indexing during the database design phase

Indexing during the design phase of a database project generally fits a very small range of needs. There is only so much guesswork about user behavior that you can make. There are specifically two situations where it is essential to define indexes in your design:

- **Uniqueness Constraints** PRIMARY KEY and UNIQUE constraints automatically create an index.
- **Foreign Key Columns** Columns that reference a FOREIGN KEY constraint are often, but not always a likely target for an index.

Let's explore these two situations.

Uniqueness Constraints

In Skill 1.1, we created PRIMARY KEY constraints on all of the tables in our design. PRIMARY KEY constraints are enforced by creating a unique index on the columns to speed the search for the duplicated value, plus a unique index does not allow duplicated data. By default, they create a unique clustered index (where the leaf pages of the B-Tree structure are the actual data pages as opposed to just pointers to the data pages), but there are situations where the clustered index is best served on a different column (This will be covered later in "Implement clustered index columns by using best practices").

As an example, consider the following table structure:

Click here to view code image

```
CREATE TABLE Examples.UniquenessConstraint
(
    PrimaryUniqueValue int NOT NULL,
    AlternateUniqueValue1 int NULL,
    AlternateUniqueValue2 int NULL
);
```

When you have a value that you need to be the primary key value, you can use a PRIMARY KEY constraint. So, using ALTER TABLE (or inline as part of the initial

CREATE TABLE statement) you can add:

Click here to view code image

```
ALTER TABLE Examples.UniquenessConstraint
    ADD CONSTRAINT PKUniquenessContraint PRIMARY KEY
(PrimaryUniqueValue);
```

A PRIMARY KEY constraint cannot be placed on a column that allows NULL values, and you get an error if you try (or in a CREATE TABLE statement it sets the column to not allow NULL values). In cases where you have alternate columns that are used to identify a row (typical when you use an artificial surrogate value like a meaningless integer for a primary key, which is covered in Chapter 2 in more detail, you can add a UNIQUE constraint that can be placed on columns that allow NULL values, something demonstrated later in this section):

Click here to view code image

```
ALTER TABLE Examples.UniquenessConstraint
    ADD CONSTRAINT AKUniquenessContraint UNIQUE
          (AlternateUniqueValue1, AlternateUniqueValue2);
```

The uniqueness constraints created indexes with the same name as the constraints behind the scenes, which you can see in sys.indexes:

Click here to view code image

```
SELECT type_desc, is_primary_key, is_unique,
is_unique_constraint
FROM   sys.indexes
WHERE  OBJECT_ID('Examples.UniquenessConstraint') =
object_id;
```

This shows you that the index is clustered, and is unique.

Click here to view code image

```
name                 type_desc       is_primary_key
is_unique is_unique_constraint
-------------------- --------------- -------------- -------
-- ------------------
PKUniquenessContraint
CLUSTERED       1               1          0
AKUniquenessContraint
NONCLUSTERED    0               1          1
```

When you have constraints on all of the data that needs to be unique for an OLTP database, you often have a large percentage of the indexes you need. OLTP databases are generally characterized by short transactions and simple queries, usually looking for one row (even if the query sometimes looks for a range of data because the user doesn't know

how to spell a given value.)

In Chapter 4, Skill 4.1, we discuss optimizing indexes, including how to determine if indexes are being used. However, indexes that are created by uniqueness constraints should not be considered for removal. Even if the index is never used to improve the performance of a query, it is essential to your data integrity to make sure that if a value is supposed to be unique, that a constraint ensures that values are unique. NULL values behave differently in UNIQUE indexes than in almost any other place in SQL Server. A PRIMARY KEY constraint does not allow any NULL values in columns, but a UNIQUE constraint and a unique index does. So, using the table we created, if we try creating the following rows:

Click here to view code image

```
INSERT INTO Examples.UniquenessConstraint
            (PrimaryUniqueValue, AlternateUniqueValue1,
AlternateUniqueValue2)
VALUES (1, NULL, NULL), (2, NULL, NULL);
```

We then receive the following error message:

Click here to view code image

```
Msg 2627, Level 14, State 1, Line 95
Violation of UNIQUE KEY constraint 'AKUniquenessContraint'.
Cannot insert duplicate key
in object 'Examples.UniquenessConstraint'. The duplicate key
value is (<NULL>, <NULL>)
```

What is initially confusing about this is that we said earlier that NULL never was equal to NULL. This is still true, but in the index keys, two NULL values are treated as duplicate missing values.

Foreign Key Columns

When implementing a FOREIGN KEY constraint, it is generally a good idea to index the key columns in the referencing tables. For example, consider the following three tables:

Click here to view code image

```
--Represents an order a person makes, there are 10,000,000 +
rows in this table
CREATE TABLE Examples.Invoice
(
    InvoiceId   int NOT NULL CONSTRAINT PKInvoice PRIMARY KEY,
    --Other Columns Omitted
);
--Represents a type of discount the office gives a customer,
--there are 200 rows in this table
CREATE TABLE Examples.DiscountType
(
```

```
    DiscountTypeId   int NOT NULL CONSTRAINT PKDiscountType
PRIMARY KEY,
    --Other Columns Omitted
)
--Represents the individual items that a customer has ordered,
There is an average of
--3 items ordered per invoice, so there are over 30,000,000
rows in this table
CREATE TABLE Examples.InvoiceLineItem
(
   InvoiceLineItemId int NOT NULL CONSTRAINT PKInvoiceLineItem
PRIMARY KEY,
   InvoiceId int NOT NULL
          CONSTRAINT FKInvoiceLineItem$Ref$Invoice
                REFERENCES Examples.Invoice (InvoiceId),
   DiscountTypeId int NOT NULL
          CONSTRAINT FKInvoiceLineItem$Ref$DiscountType
                REFERENCES Examples.DiscountType
(DiscountTypeId)
    --Other Columns Omitted
);
```

There are two foreign key columns in the InvoiceLineItem table to cover. The InvoiceId column has mostly unique values, with an average of 3 rows per invoice. It is also a typical thing a user might do; grabbing all of the invoice items for an invoice. Hence, that is a column that almost certainly benefits from an index (and as we discuss later in the section "Implement clustered index columns by using best practices", perhaps even a clustered index if the reference is used frequently enough.) Create that index as a non-clustered index for now as:

Click here to view code image

```
CREATE INDEX InvoiceId ON Examples.InvoiceLineItem (InvoiceId);
```

When creating an index on a not-very selective column, like perhaps the DiscountTypeId, where out of 30 million rows, there are just 20 distinct values in 100000 rows. This column could benefit from a filtered index, which is an index that has a WHERE clause. So if almost all rows were NULL, it could be that searching for a row with the value of NULL in the index would be useless. However, a search for other values actually could use the index. So you could create a filtered index as:

Click here to view code image

```
CREATE INDEX DiscountTypeId ON
Examples.InvoiceLineItem(DiscountTypeId)
                                                WHERE DiscountTypeId
IS NOT NULL;
```

Filtered indexes can have any columns in the WHERE clause, even if not represented in the index keys or included columns (which we use later in this chapter in the section: "Distinguish between indexed columns and included columns").

When creating an INDEX, if the data in the key columns is always unique (such as if the columns of the index are a superset of a UNIQUE and/or PRIMARY KEY constraint columns,) declare the index as UNIQUE, as in:

Click here to view code image

```
CREATE UNIQUE INDEX InvoiceColumns ON
Examples.InvoiceLineItem(InvoiceId,
```

It is typically desirable to have indexes that enforce uniqueness to be based on a constraint, but this is not a requirement. Any UNIQUE index, even a filtered one, can be declared as only allowing unique values which will disallow duplicated index key values.

Need More Review The CREATE INDEX Statement

There are many other settings in the `CREATE INDEX` statement that is useful to understand, that are covered in more detail on the MSDN site: *https://msdn.microsoft.com/en-us/library/ms188783.aspx*.

In the WideWorldImporters database, there are indexes on all of the foreign keys that were generated when creating that database. Beyond guessing what one can do to decide whether an index would be useful, it is essential to understand a query plan. Most of the figures in this chapter are query plans to demonstrate what is going on in the query optimizer and query processor. As an example, use one of the relationships in the WideWorldImporters database, between the Sales.CustomerTransactions and Application.PaymentMethods tables.

Note Accessing the Sample Database

To follow along with the examples in this chapter, and later ones that use the Microsoft sample database: `WideWorldImporters`, you can get this database at the following address: *https://github.com/Microsoft/sql-server-samples/releases/tag/wide-world-importers-v1.0*. It will be used throughout the book when we need a database that is pre-loaded with data.

In the Sales.CustomerTransactions table, there are 97147 rows. The index on the foreign key column is non-clustered, so every use of the non-clustered index requires a probe of the clustered index to fetch the data (referred to as a *bookmark lookup*), so it is very unlikely the index is used for a predicate. Take a look at the data in the PaymentMethodId column:

Click here to view code image

```
SELECT PaymentMethodId, COUNT(*) AS NumRows
FROM   Sales.CustomerTransactions
GROUP  BY PaymentMethodID;
```

You can see that there are just two values in use:

Click here to view code image

```
PaymentMethodId NumRows
--------------- -----------
4               26637
NULL            70510
```

Take a look at the plan of the following query that the system might perform, searching for CustomerTransactions rows where the PaymentMethodId = 4:

Click here to view code image

```
SELECT *
FROM   Sales.CustomerTransactions
WHERE PaymentMethodID = 4;
```

This returns the expected 26637 rows, and has the actual plan shown in Figure 1-2. The Compute Scalar operator is there because we returned all columns, and there is a computed column in the table named IsFinalized.

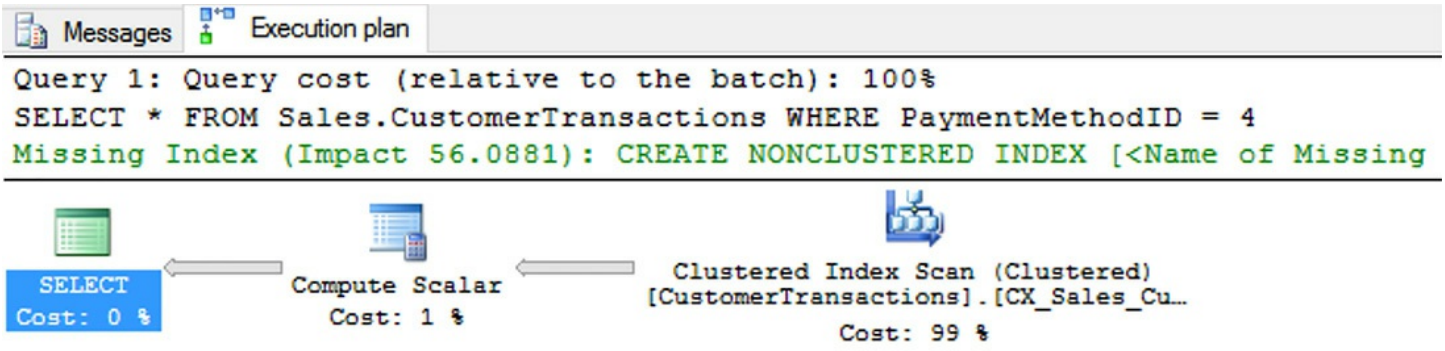


FIGURE 1-2 The plan from the query for PaymentMethodId = 4

There are three ways using the GUI to get the query plan. From the Query Menu, select:

1. **Display Estimated Plan** This shows you the plan that is likely to be used to perform the query. The plan can change when the query is performed, due to many factors such as query load, memory available, etc. All row counts and costs are guesses based on the statistics of the index, and it does not require the query to be performed. Whether or not parallelism can be used is determined during execution based on system settings (such as the sp_configure setting 'cost threshold for parallelism' and 'max degree of parallelism') and the load on the system at execution time.
2. **Include Actual Execution Plan** This represents the plan that is used, including

actual row counts, use of parallelism, etc. You get the actual plan after the query has completed in its entirety.

3. **Include Live Query Statistics** When you are working with a complex, long-running query, you can see data moving through the actual query plan operators live. It can help you diagnose issues with a large query by letting you see the problem spots in real time.

> **Note Textual plan Options**
>
> Additionally, there are several ways to get a textual plan when you need it. Two examples are SET SHOWPLAN_TEXT to get the estimated plan, and SET STATISTICS PROFILE to get the actual query plan.

For now, we ignore the Missing Index listed in the plan (and for future cases edit them out until we get to the section on "Included Columns"), but the point here is that the index was not used. However, it is important to note that while an index is not generally useful, there are scenarios where the index actually turns out to be useful:

- If the only column that was returned from the query was the PaymentMethodId since all of the data is there in the index, then it is useful.
- An index is also useful when you are searching for a value that does not exist in the table. The statistics of the index do not tell the optimizer that no rows are returned from the query, only that very few are returned, so using the index should be fast enough. We review managing statistics in more detail in Chapter 4, but they are basically structures that help the optimizer to guess how many rows are returned by a given query based on a sampling of the data at a given point in time.

These scenarios are why foreign key indexes are often applied to all foreign key columns, even if the indexes applied are not generally useful.

> **Need More Review? Deeper dive into indexing**
>
> Indexing is a complex topic that we only review in some of the primary scenarios. For more information, a great resource is "Expert Performance Indexes in SQL Server" from Apress by Grant Fritchey and Jason Strate (*http://www.apress.com/9781484211199*).

Indexing once data is in your tables

Although the indexing you might do to your tables before adding data is essentially part of the structure, and the rest of the indexes are strictly intended to improve performance. In this section, we cover several scenarios to consider when adding indexes to tables. Some of these scenarios crop up during development, even when you have very little data in your tables. Some do not show up until the data grows during performance testing or production

loads. All of Chapter 4 delves more into the ongoing tuning of your system, but for now we look at some common query types to tune, no matter how you discover the need.

- Common search paths
- Joins
- Sorting data

Unless you have very simplistic needs, it is hard to know exactly how queries behave in a real scenario, so in most cases it is better to test out your expectations rather than guess about performance.

Exam Tip

While tuning a real database should generally be done with real data, seeing real needs, this is not the case for the exam. The situations more likely follow a very deliberate pattern similar to the ones we discuss in the next sections. The upcoming examples are not exhaustive as there are many different scenarios that can use an index to improve performance.

Common search paths discovered during development

The process of adding indexes starts during the development phase of the project. Even with smaller amounts of data in a table, there are given access paths that do not correspond exactly to the indexes the uniqueness constraints you have started with added. For example, in the WideWorldImporters database, in the Sales.Orders table, the CustomerPurchaseOrderNumber is not a key value (there do exist duplicated values, because the purchase order number can be duplicated for different, or even the same customer in this design). During design, it was not a foreign key, nor was it a key in a uniqueness constraint. When the application was created, it included the following query:

Click here to view code image

```
SELECT CustomerID, OrderID, OrderDate, ExpectedDeliveryDate
FROM  Sales.Orders
WHERE CustomerPurchaseOrderNumber = '16374';
```

In the base set of rows here in the WideWorldImporters database (there are methods included for adding more data to give you more data to work with), the query runs very fast, returning just 6 rows. In order to see just how well it performs, you can use two commands in Transact-SQL to see some very important statistics, that along with the query plan, give you the important information on how the query is operating.

Click here to view code image

```
SET STATISTICS TIME ON;
SET STATISTICS IO ON;
```

```
SELECT CustomerID, OrderId, OrderDate, ExpectedDeliveryDate
FROM  Sales.Orders
WHERE CustomerPurchaseOrderNumber = '16374';

SET STATISTICS TIME OFF;
SET STATISTICS IO OFF;
```

The plan returns what is shown in Figure 1-3.

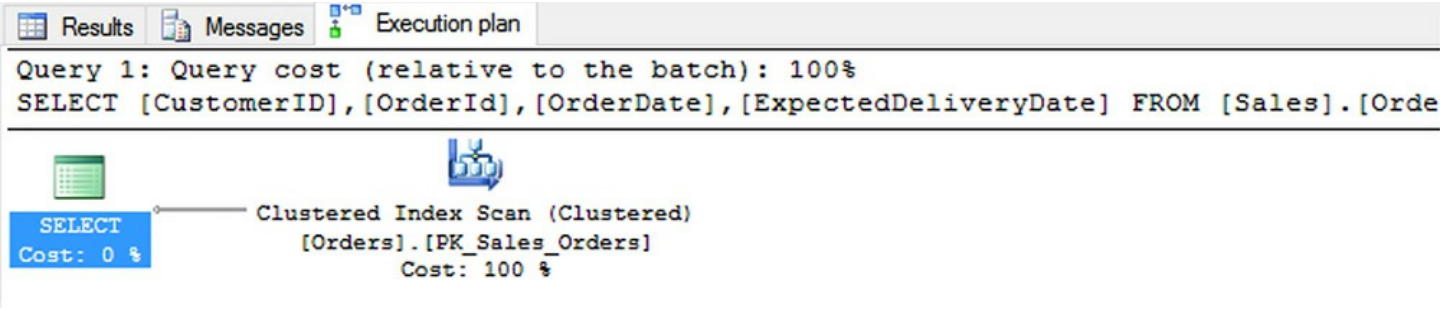


FIGURE 1-3 Query plan that does not use an index

Along with the query results, there are a few additional messages. We are reducing to the pertinent ones here in our output, but you can see the compile and parse times, and the overall execution time in addition to the following:

Click here to view code image

```
Table 'Orders'. Scan count 1, logical reads 692, physical reads
0, read-ahead reads 0,
lob logical reads 0, lob physical reads 0, lob read-ahead reads
0.

 SQL Server Execution Times:
   CPU time = 15 ms,  elapsed time = 20 ms.:
```

The query only takes around 20 milliseconds (ms) on a VM on a Surface Pro 4 with 8 GB of RAM for the VM (and 8GB for the host), scanning the table, touching all 692 pages of the table which has 73595 rows. All of the pages were in RAM already, so there are no physical reads (very common when testing individual queries and there is no memory pressure. You can clear the cache using DBCC DROPCLEANBUFFERS, but the most important number for indexing is the logical reads. Consistent readings of a large number of physical reads are more indicative of not enough RAM to cache data). However, if this is a table to which data is being actively written, scanning those 692 pages means that every single row is touched, and therefore locked in on-disk tables at some point in time, causing concurrency issues that are covered in more detail in Chapter 3, "Manage Database Concurrency."

Next, add an index to the Sales.Orders table on the CustomerPurchaseOrderNumber column, to attempt to speed the query:

Click here to view code image

```
CREATE INDEX CustomerPurchaseOrderNumber ON
Sales.Orders(CustomerPurchaseOrderNumber);
```

> **Note Our sample database**
>
> The examples use tables from the `WideWorldImporters` database to review different types of indexing utilization. If you desire to try the queries yourself to make the same changes, make sure that you are working on your own copy of this database before making changes that affect other users.

Now, perform the same query on CustomerPurchaseOrderNumber = '16374', and the following query plan is used, as shown in Figure 1-4.

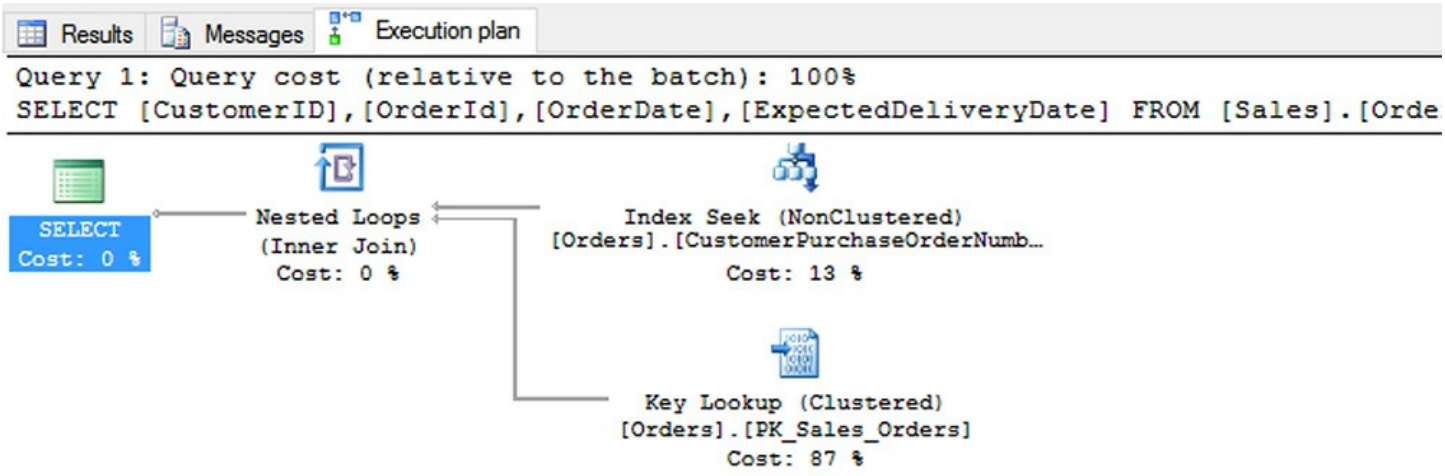


FIGURE 1-4 Query plan after adding an index on the `CustomerPurchaseOrderNumber`

The query plan looks more complex. There is a join, even though the query uses a single table. SQL Server now uses the index-seek operation to find the six matching rows, but all it has are the CustomerID and the OrderID from the index keys. So it needs to use a JOIN operator to join to the clustered index to get the rest of the data. While the plan is more complex, the results are a lot better statistically, as you can see:

Click here to view code image

```
Table 'Orders'. Scan count 1, logical reads 20, physical reads
0, read-ahead reads 0,
lob logical reads 0, lob physical reads 0, lob read-ahead reads
0.
 SQL Server Execution Times:
   CPU time = 0 ms,  elapsed time = 0 ms.
```

It took only 20 logical reads, and less than 1 millisecond to perform. The reduction in 672 reads, means 672 less physical resources touched, and locked by default. As a result,

it is very useful to check out all of the queries that are used by your applications, either (ideally) from stored procedures, or as ad-hoc queries performed from your external interfaces.

Note that you can index a computed column as long as it is deterministic. You can tell if a column can be indexed, even if it is computed by using the COLUMNPROPERTYEX() function:

Click here to view code image

```
SELECT CONCAT(OBJECT_SCHEMA_NAME(object_id), '.',
OBJECT_NAME(object_id)) AS TableName,
       name AS ColumnName, COLUMNPROPERTYEX(object_id, name,
'IsIndexable') AS Indexable
FROM   sys.columns
WHERE is_computed = 1;
```

Search conditions are typically the most obvious to index because the affect people directly. When a user searches on an unindexed column in a large table (relative to hardware capabilities), you may see locking, blocking, or using some settings (such as the database setting READ COMMITTED SNAPSHOT), high tempdb utilization. The needs are more random that in the following situation we will cover.

Joins

While simple index needs often manifest themselves as table scans, when joining data in two tables, the need for an index instead may show up as a different join operator than a nested-loops join. Nested loops work best when one set is very small, or the cost of seeking for a row in that set is inexpensive. It works by going row by row in one of the inputs, and seeking for a matching value in the other. When the cost of seeking in both sets is too costly, a Hash Match operator is used. This operator makes a pseudo hash index by segmenting values into buckets of values that can be easier to scan using a hash function. It does not need any order to the operation, so it can work to join two really large sets together.

As an example, drop the foreign key index from the Sales.Orders table named FK_Sales_Orders_ContactPersonID using the following command:

Click here to view code image

```
DROP INDEX FK_Sales_Orders_ContactPersonID ON Sales.Orders;
```

Now, search for the Sales.Orders rows for any person with a preferred name of ‘Aakriti:’

Click here to view code image

```
SELECT OrderId, OrderDate, ExpectedDeliveryDate,
People.FullName
```

```
FROM  Sales.Orders
        JOIN Application.People
            ON People.PersonID = Orders.ContactPersonID
WHERE  People.PreferredName = 'Aakriti';
```

The PreferredName column is not indexed. Figure 1-5 shows the actual query plan, along with the typical query stats output.

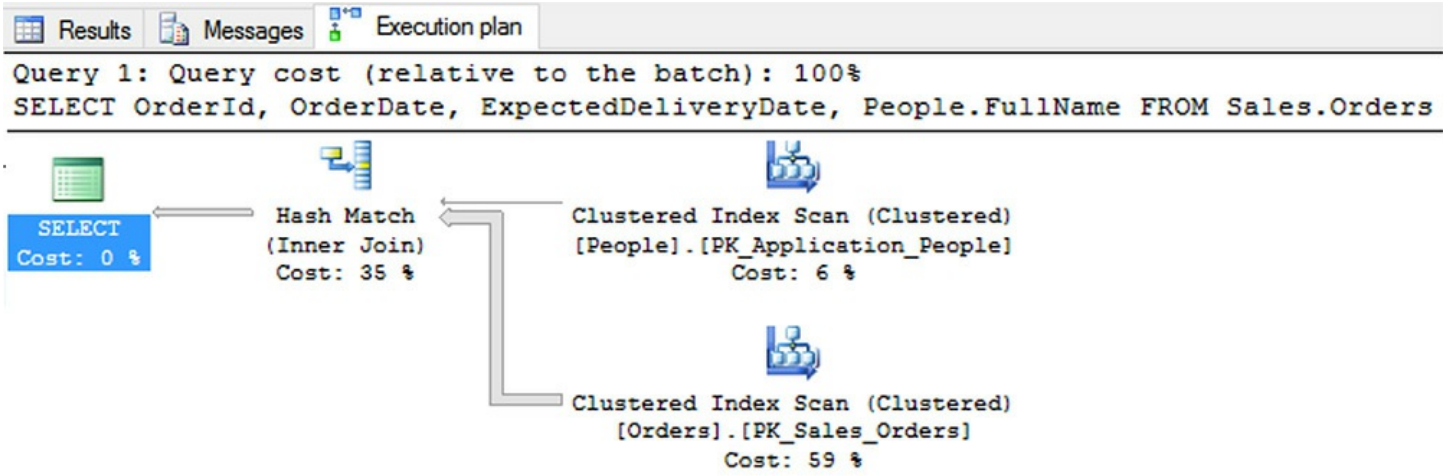


FIGURE 1-5 Query plan and statistic output for unindexed foreign key index in join

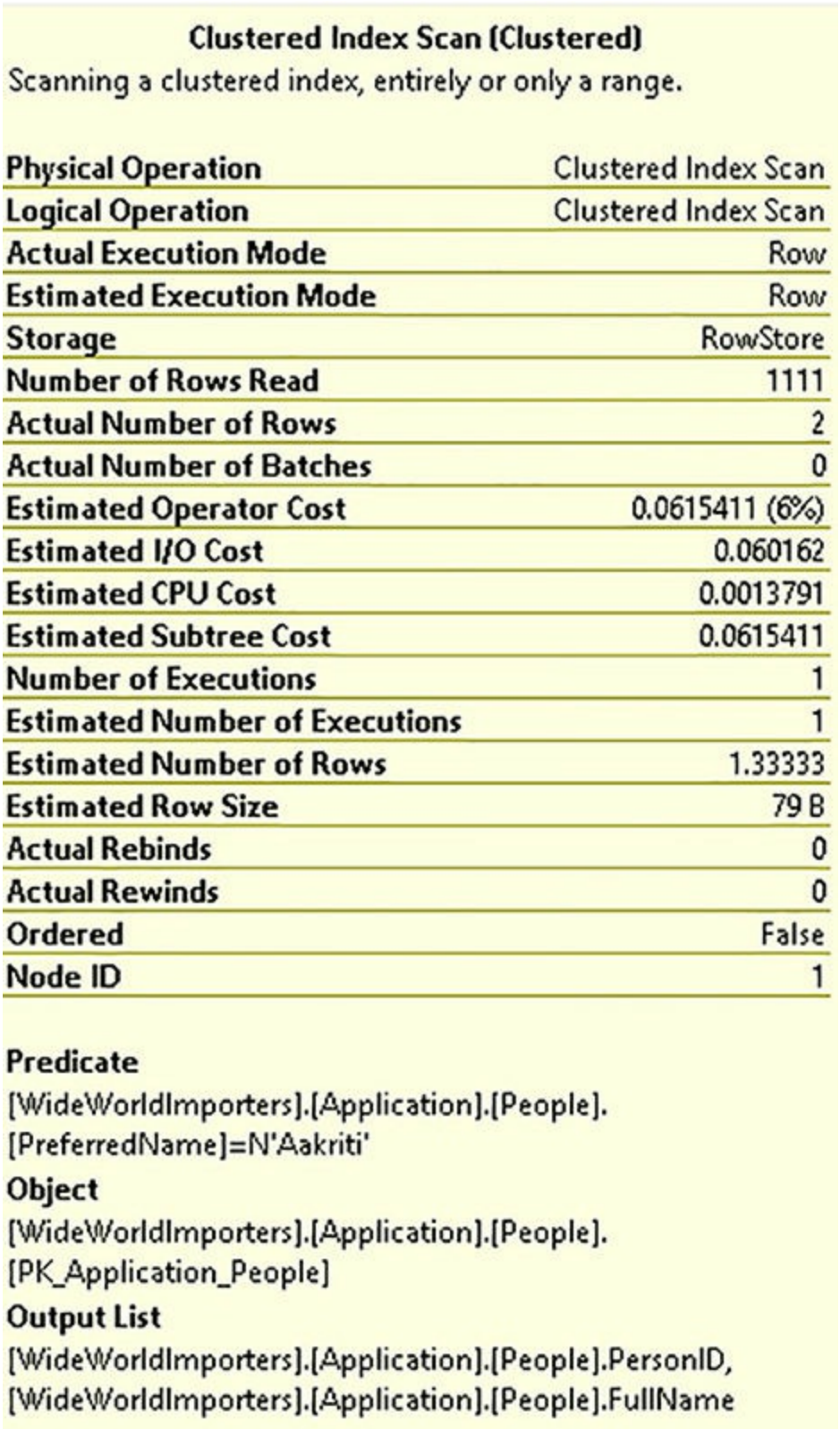
Figure 1-5 has the following output:

Click here to view code image

```
Table 'Workfile'. Scan count 0, logical reads 0, physical reads
0, read-ahead reads 0,
lob logical reads 0, lob physical reads 0, lob read-ahead reads
0.

Table 'Worktable'. Scan count 0, logical reads 0, physical
reads 0, read-ahead reads 0,
lob logical reads 0, lob physical reads 0,

lob read-ahead reads 0.

Table 'Orders'. Scan count 1, logical reads 692, physical reads
0, read-ahead reads 0,
lob logical reads 0, lob physical reads 0,

lob read-ahead reads 0.

Table 'People'. Scan count 1, logical reads 80, physical reads
0, read-ahead reads 0,
lob logical reads 0, lob physical reads 0,

lob read-ahead reads 0.
```

```
CPU time = 15 ms,  elapsed time = 53 ms.
```

Hovering your mouse over the Clustered Index Scan operator for the PK_Application_People index (the clustered index on the table), you see (as depicted in Figure 1-6) the costs, but also that the predicate of PreferredName = 'Aakriti' is handled as part of this scan.



Clustered Index Scan (Clustered)

Scanning a clustered index, entirely or only a range.

Physical Operation	Clustered Index Scan
Logical Operation	Clustered Index Scan
Actual Execution Mode	Row
Estimated Execution Mode	Row
Storage	RowStore
Number of Rows Read	1111
Actual Number of Rows	2
Actual Number of Batches	0
Estimated Operator Cost	0.0615411 (6%)
Estimated I/O Cost	0.060162
Estimated CPU Cost	0.0013791
Estimated Subtree Cost	0.0615411
Number of Executions	1
Estimated Number of Executions	1
Estimated Number of Rows	1.33333
Estimated Row Size	79 B
Actual Rebinds	0
Actual Rewinds	0
Ordered	False
Node ID	1

Predicate

[WideWorldImporters].[Application].[People].
[PreferredName]=N'Aakriti'

Object

[WideWorldImporters].[Application].[People].
[PK_Application_People]

Output List

[WideWorldImporters].[Application].[People].PersonID,
[WideWorldImporters].[Application].[People].FullName

FIGURE 1-6 Operator costs for the Clustered Index Scan operator for the

PK_Application_People index

As you can see, the query optimizer scans the two indexes, and the Hash Match operator builds a hash index structure, and then matches the rows together. Adding back the index on the foreign key columns:

Click here to view code image

```
CREATE INDEX FK_Sales_Orders_ContactPersonID ON Sales.Orders
--Note that USERDATA is a filegroup where the index was
originally
                (ContactPersonID ASC ) ON USERDATA;
```

Executing the query again shows a better result, though not tremendously, as shown in Figure 1-7.

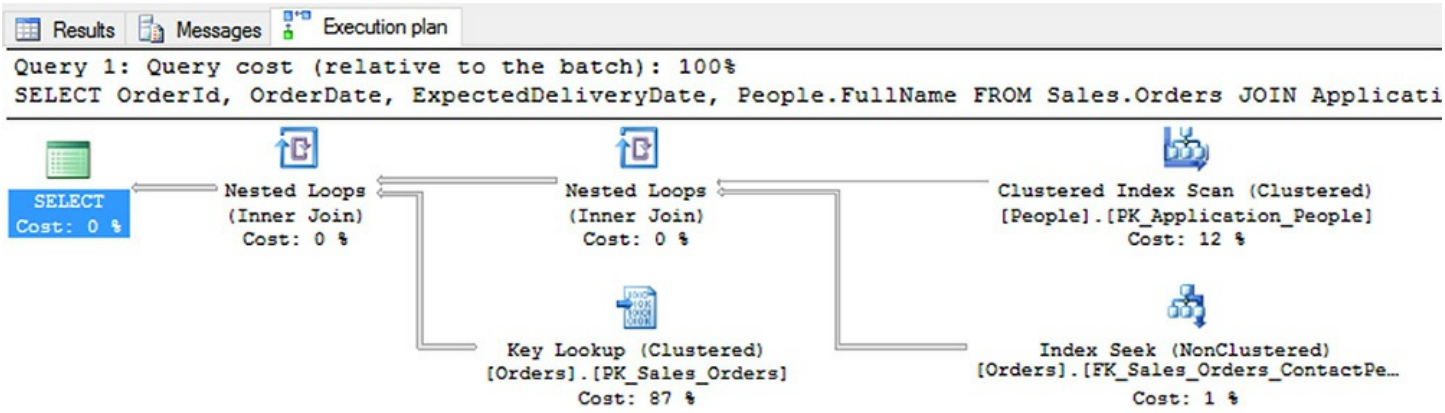


FIGURE 1-7 Query plan after adding back the foreign key index

Figure 1-7 has the following output:

Click here to view code image

```
Table 'Orders'. Scan count 2, logical reads 695, physical reads
0, read-ahead reads 0,
lob logical reads 0, lob physical reads 0,

lob read-ahead reads 0.

Table 'People'. Scan count 1, logical reads 80, physical reads
0, read-ahead reads 0,
lob logical reads 0, lob physical reads 0,

lob read-ahead reads 0.

CPU time = 0 ms,  elapsed time = 17 ms.
```

The big cost here is the Key Lookup operator to fetch the rest of the Sales.Orders columns in our query. This cost is what the missing index hint has been suggesting for nearly every query, and is the topic of the next section of this chapter. The query can be

improved upon one more time by indexing the PreferredName column, so the query processor doesn't have to test every single row in the Application.People table to see if it matches PreferredName = 'Aakriti'.

Click here to view code image

```
CREATE INDEX PreferredName ON Application.People
(PreferredName) ON USERDATA;
```

Finally, perform the query again to see the plan and statistics shown in Figure 1-8.

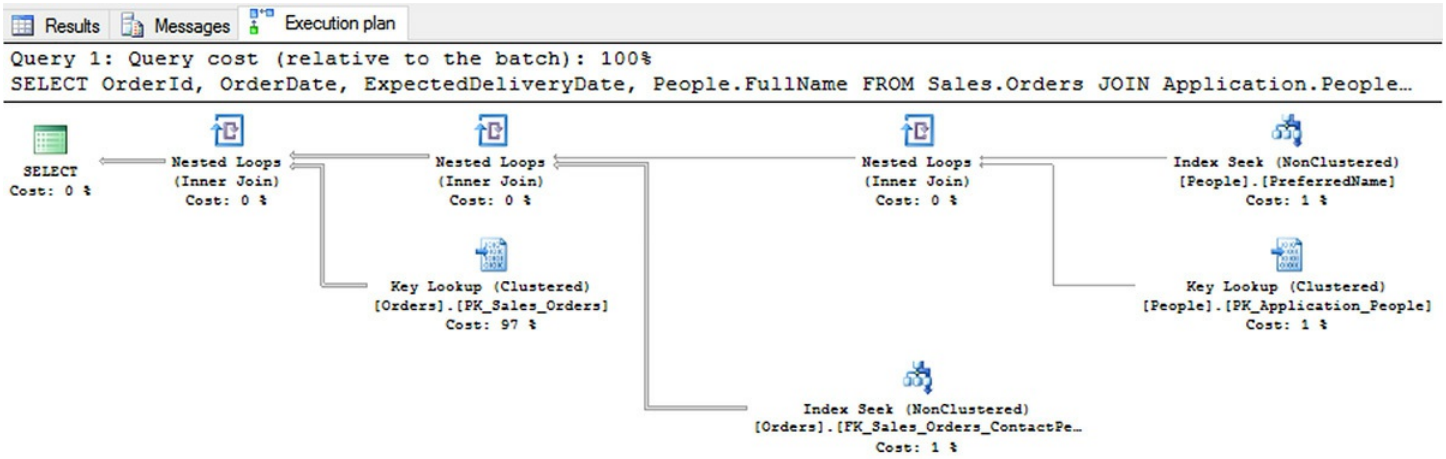


FIGURE 1-8 Query plan after adding index on `Application.People.PreferredName`

Figure 1-8 has the following output:

Click here to view code image

```
Table 'Orders'. Scan count 2, logical reads 695, physical reads
0, read-ahead reads 0,
lob logical reads 0, lob physical reads 0,

lob read-ahead reads 0.

Table 'People'. Scan count 1, logical reads 6, physical reads
0, read-ahead reads 0, lob
logical reads 0, lob physical reads 0, lob read-ahead reads 0.

CPU time = 0 ms,  elapsed time = 19 ms.
```

This is not a tremendous improvement, and is just 74 less accessed pages, and execution times are typically the same. Generally speaking though, the fewer pages read in the process of executing the query, the better, particularly as the number of queries increase in an active system.

Note the Key Lookup operator that is 97 percent of the cost of this query. In a following section on included columns, we review how to erase that cost, and lower the logical reads to very few.

Sorts

The final query situation we look at is sorts. When you need to sort data, either for an ORDER BY clause or for some operation in the query where sorting data would make query operation quicker (the last join operator that we haven't mentioned yet, called the Merge Join operator, requires sorted inputs to match rows from one large input set to another large set, in a quicker manner than using the Hash Merge algorithm previously mentioned).

> **Note Indexing and sorting**
>
> The examples in this section use only columns that show up in the operation in order to show how indexing and sorting work together, and it eliminates some of the costs of the bookmark lookup. The next section examines this phenomenon in more detail.

A part of the CREATE INDEX statement we have not yet looked at is sorting of the index keys, particularly useful with composite index keys. By default, the data in the index is sorted in ascending order, so the indexes created so far have been ascending by default. The query processor can scan the index in either direction, so for a *simple index* (one with a single key column), this is generally not a problem. For *composite indexes* (those with greater than a single key column) it can be an issue.

As an example, consider the following query of the entire Sales.Orders table, sorted in SalespersonPersonID and OrderDate order. Both are explicitly spelled out as ASC, meaning ascending, which is the default. Note too that we only return the columns that are being sorted on to make the example simpler.

Click here to view code image

```
SELECT SalespersonPersonId, OrderDate
FROM Sales.Orders
ORDER BY SalespersonPersonId ASC, OrderDate ASC;
```

Figure 1-9 shows the plan, which includes a scan through the data, a sort, and it even shows that the query used parallelism, since we're running on a VM with 2 CPUs allocated. In other words, this was not a trivial query.

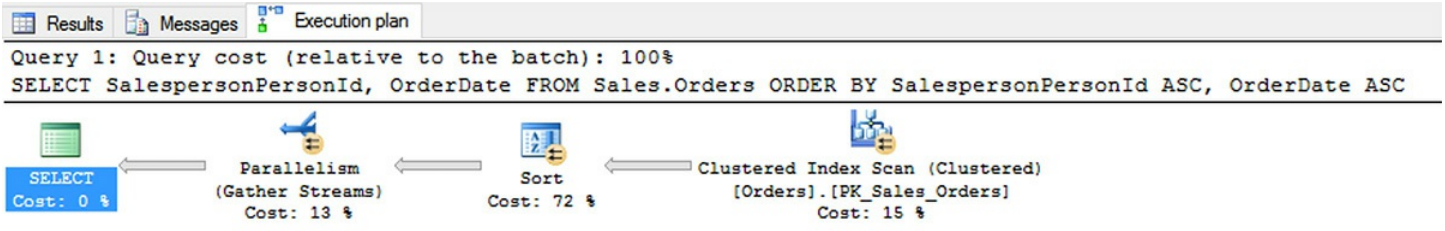


FIGURE 1-9 Sorting results prior to adding index

Figure 1-9 has the following output:

Click here to view code image

```
Table 'Worktable'. Scan count 0, logical reads 0, physical
reads 0, read-ahead reads 0,
lob logical reads 0, lob physical reads 0,

lob read-ahead reads 0.

Table 'Orders'. Scan count 3, logical reads 758, physical reads
0, read-ahead reads 0,
lob logical reads 0, lob physical reads 0,

lob read-ahead reads 0.

CPU time = 94 ms,  elapsed time = 367 ms.
```

Now, add an index to support this query, as we know that this query is performed very often in our enterprise. Add a composite index, and explicitly show that we are sorting the keys in ascending order, for the query:

Click here to view code image

```
CREATE INDEX SalespersonPersonID_OrderDate ON Sales.Orders
                                (SalespersonPersonID ASC,
OrderDate ASC);
```

Perform the query just as we did in the first attempt. Figure 1-10, shows that the plan has changed, as now it can get the data in a pre-sorted manner, with the primary cost of the query now embedded in the bookmark lookup.

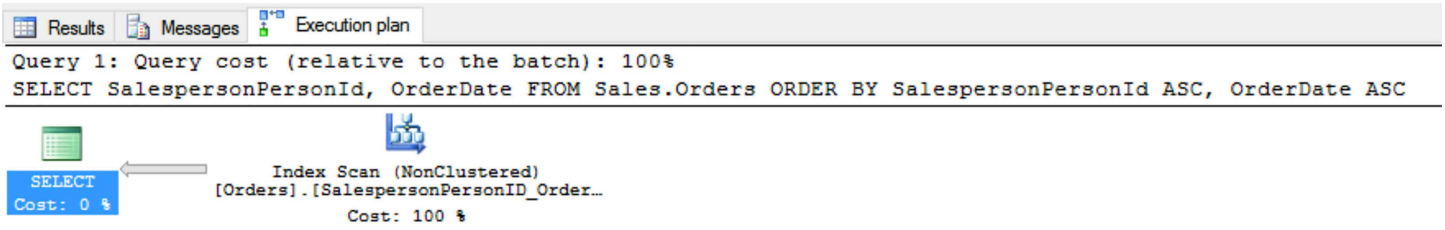


FIGURE 1-10 Query plan after adding the index to the table

Figure 1-10 has the following output:

Click here to view code image

```
Table 'Orders'. Scan count 1, logical reads 157, physical reads
0, read-ahead reads 0,
lob logical reads 0, lob physical reads 0, lob read-ahead reads
0.

CPU time = 47 ms,  elapsed time = 217 ms.
```

If the order you request is completely opposite of how the index is sorted, you will find

that nothing in the plan will change:

Click here to view code image

```
SELECT SalespersonPersonId, OrderDate
FROM Sales.Orders
ORDER BY SalespersonPersonId DESC, OrderDate DESC;
```

If your sorting needs don't match the index exactly, it's still useful to the query, but only until there is a mismatch. For example, change the ORDER BY to either of the following (DESC is descending):

- ORDER BY SalespersonPersonId DESC, OrderDate ASC;
- ORDER BY SalespersonPersonId ASC, OrderDate DESC;

And you see the plan changes to what is shown in Figure 1-11.

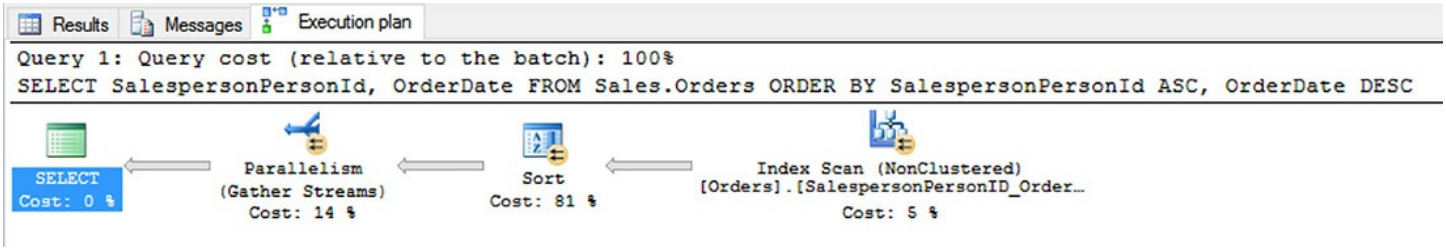


FIGURE 1-11 Query plan when the sort order does not match

The query processor was able to skip sorting the data based on the first column by using the index, but then it had to sort the second column using a separate operator, rather than just scanning the data in order. As such, it is important to note the order of the columns in the ORDER BY clause, if you were given a question matching the index with the ORDER BY clause.

One place where sorting often is helped by indexes is when joining two large sets. The query plan can use a Merge Join operator to join two sorted sets together, by matching item after item, since they are in sorted order. As an example, take a join of two tables, the Sales.Orders and the Application.People, returning all of the rows in the tables, but just their key values:

Click here to view code image

```
SELECT Orders.ContactPersonID, People.PersonID
FROM   Sales.Orders
         INNER JOIN Application.People
            ON Orders.ContactPersonID = People.PersonID;
```

Executing this, you see that since there is an index on the foreign key column in Sales.Orders, and the PRIMARY KEY constraint on the Application.People table, the data is sorted, so it can use a Merge Join operator, as seen in Figure 1-12.

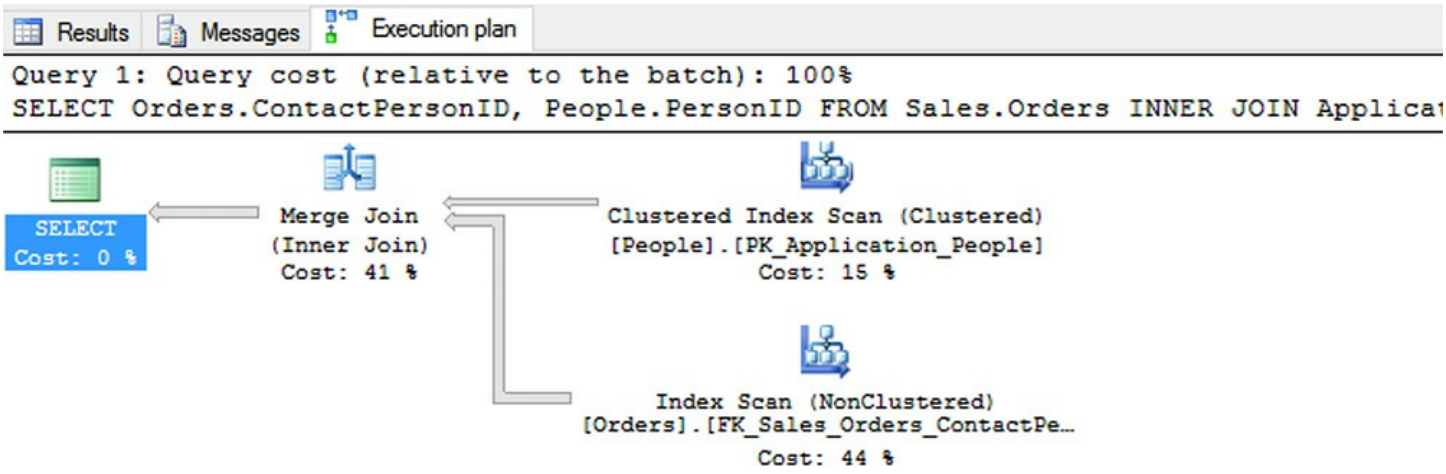


FIGURE 1-12 Merge Join operator due to large output and sorted inputs

To be entirely fair, the output of this query is nonsensical because it returns two columns that are equivalent in all 72,595 rows. However, when you are doing joins between multiple tables, you often see a Merge Join operator appear in plans when nothing but indexed columns are accessed from the tables that are being joined.

Carefully consider how you use non-clustered indexes to support sorts, as the cost of the bookmark lookup often tips the plan towards using a scan of the base structure.

Distinguish between indexed columns and included columns

When fetching only a few rows (as you generally do when you are querying an OLTP database), the overhead of this lookup is not terribly costly. It requires reading two or three extra pages in the clustered index, but this cost is extremely minimal compared to reading every physical page of data for an entire table.

However, as the number of rows you return grows, the bookmark lookup operations become more and more of a drag on performance. When you need to run a query that returns a lot of rows, but doesn't need all of the data in the table, there is a feature known as *included columns* that allows you to use an index to *cover* the entire needs of the query. When an index has all of the data that is needed to return the results of a query, either in the key columns, or included columns, it is referred to as a *covering index* for a query.

As an example, take a look back at this query we have used previously:

Click here to view code image

```
SELECT OrderId, OrderDate, ExpectedDeliveryDate,
People.FullName
FROM  Sales.Orders
        JOIN Application.People
            ON People.PersonID = Orders.ContactPersonID
WHERE  People.PreferredName = 'Aakriti';
```

Remember back in Figure 1-11, this query was very efficient in terms of finding the rows

that needed to be returned in the Sales.Orders table, but had one operator that was 97 percent of the cost of execution, and required 695 pages to be read in the Sales.Orders table.

Now perform the query, and see the plan that is output. In the plan shown in Figure 1-13, the Key Lookup operator is 97% of the overall cost of the query. There are two Key Lookup operators in the plan, so remove both of them in the simplest case.

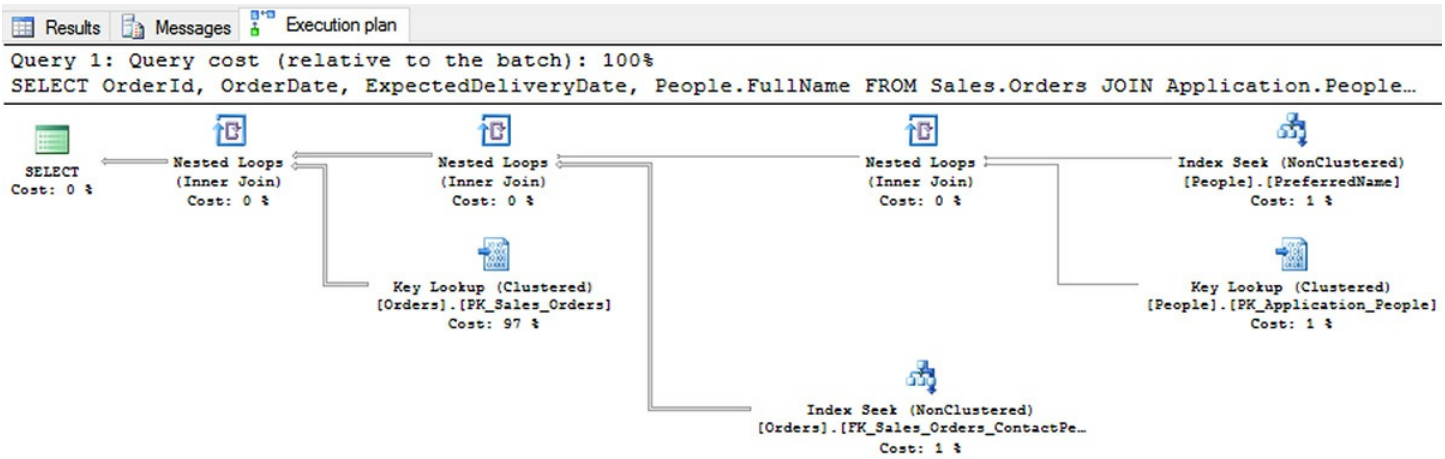


FIGURE 1-13 Query plan with a very high costs for the Key Lookup operators of 98%

Figure 1-13 has the following output:

Click here to view code image

```
Table 'Orders'. Scan count 2, logical reads 695, physical reads
0, read-ahead reads 0,
lob logical reads 0, lob physical reads 0,

lob read-ahead reads 0.

Table 'People'. Scan count 1, logical reads 6, physical reads
0, read-ahead reads 0, lob
logical reads 0, lob physical reads 0,

lob read-ahead reads 0.

CPU time = 0 ms,  elapsed time = 141 ms.
```

As a first step, if we simplify our query to just use the columns that are indexed, the Sales.Orders.ContactPersonID column from the foreign key index that was created by the database designer, and the Application.People.PreferredName column (which also includes the PersonId since it is the clustering key), you see that all of the data you need for your query (for all clauses, SELECT, FROM, WHERE, etc) can be found in the index keys. Executing the query:

Click here to view code image

```
SELECT Orders.ContactPersonId, People.PreferredName
```

```
FROM  Sales.Orders
        JOIN Application.People
            ON People.PersonID = Orders.ContactPersonID
WHERE  People.PreferredName = 'Aakriti';
```

Now the query plan looks wonderful, and the number of logical reads are down dramatically, as you can see in Figure 1-14. The indexes that are being sought are covering the query processor's needs. There is only one small problem. The query results are not even vaguely what the customer needs.

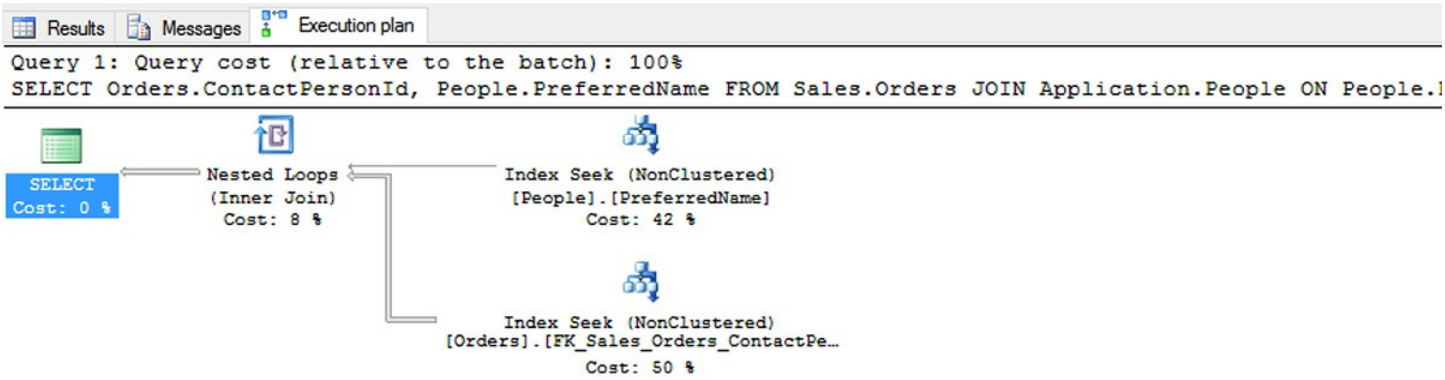


FIGURE 1-14 The Key Lookup operators have been eliminated from the plan

Figure 1-14 has the following output:

Click here to view code image

```
Table 'Orders'. Scan count 2, logical reads 4, physical reads
0, read-ahead reads 0, lob
logical reads 0, lob physical reads 0,

lob read-ahead reads 0.

Table 'People'. Scan count 1, logical reads 2, physical reads
0, read-ahead reads 0, lob
logical reads 0, lob physical reads 0,

lob read-ahead reads 0.

CPU time = 0 ms,  elapsed time = 38 ms.
```

In order to keep this performance with the minimum overhead, but providing results that were requested, you can use what is referred to as a *covering index*. The leaf nodes of a non-clustered index contains the value being indexed, along with a row locator. A covering index uses the INCLUDE keyword on the CREATE INDEX statement to include additional information on the leaf nodes. You can include any data type (even the large types like nvarchar(max), though the larger the data type, the less fits on a page, or it could even overflow to multiple pages.

For our two queries, we add another index to the Sales.Orders table (since the foreign

key index came as part of the base installation) and replace the PreferredName index that is part of the original WideWorldImporters database.

Click here to view code image

```
CREATE NONCLUSTERED INDEX
ContactPersonID_Include_OrderDate_ExpectedDeliveryDate
ON Sales.Orders ( ContactPersonID )
INCLUDE ( OrderDate,ExpectedDeliveryDate)
ON USERDATA;
GO
```

And to the PreferredName index we include the column the customer wanted, the FullName column.

Click here to view code image

```
DROP INDEX PreferredName ON Application.People;
GO
CREATE NONCLUSTERED INDEX PreferredName_Include_FullName
ON Application.People (      PreferredName )
INCLUDE (FullName)
ON USERDATA;
```

Now, perform the query:

Click here to view code image

```
SELECT OrderId, OrderDate, ExpectedDeliveryDate,
People.FullName
FROM  Sales.Orders
        JOIN Application.People
            ON People.PersonID = Orders.ContactPersonID
WHERE  People.PreferredName = 'Aakriti';
```

And the plan now looks great, and returns what the customer needs. You can see the plan in Figure 1-15.

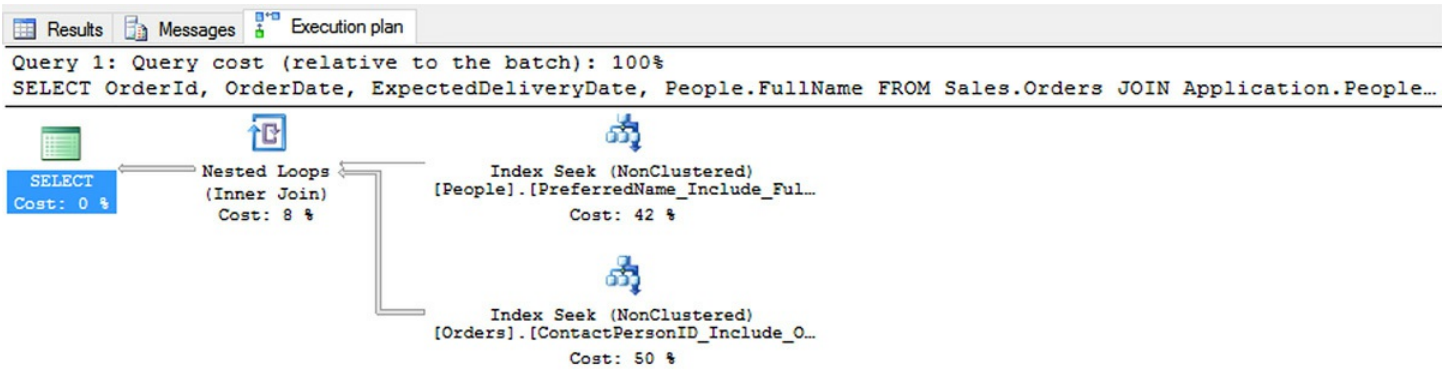


FIGURE 1-15 The query plan of the query execution after adding the covering index

Figire 1-15 has the following output:

Click here to view code image

```
Table 'Orders'. Scan count 2, logical reads 6, physical reads
0, read-ahead reads 0, lob
logical reads 0, lob physical reads 0,

lob read-ahead reads 0.

Table 'People'. Scan count 1, logical reads 2, physical reads
0, read-ahead reads 0, lob
logical reads 0, lob physical reads 0,

lob read-ahead reads 0.

CPU time = 0 ms,  elapsed time = 79 ms.
```

Covering indexes are fantastic tools for tuning queries where you are dealing with costly Key Lookup operators. However, restraint should be taken when considering whether or not to apply them. When checking the plan of a query, you are frequently given a missing index hint that encourages you to add an index with a long list of included columns. Figure 1-5 showed the plan of the following query:

```
SELECT *
```

Click here to view code image

```
FROM   Sales.CustomerTransactions
WHERE PaymentMethodID = 4;
```

When looking at the plan, there was a missing index hint as shown in Figure 1-16.

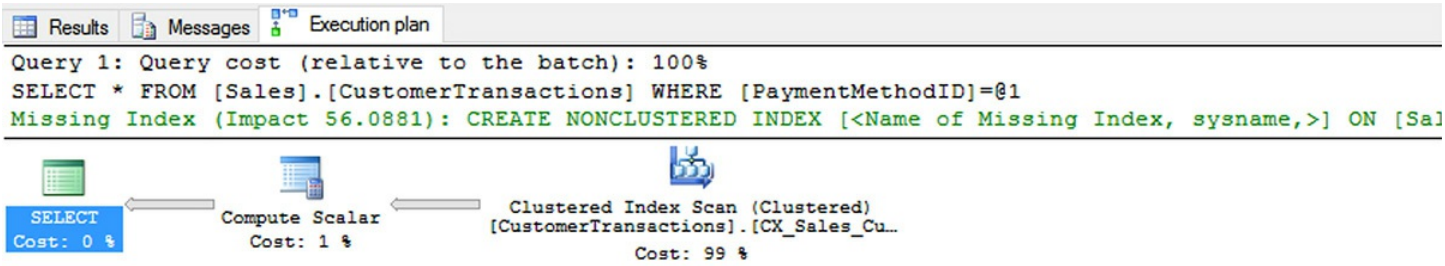


FIGURE 1-16 Showing the Missing Index hint on query plan

Hovering your cursor over the missing index shows you details in a tool-tip, or right click the plan and choose "Missing Index Details..." and you see the following index:

Click here to view code image

```
CREATE NONCLUSTERED INDEX [<Name of Missing Index, sysname,>]
ON [Sales].[CustomerTransactions] ([PaymentMethodID])
INCLUDE ([CustomerTransactionID],[CustomerID],
[TransactionTypeID], [InvoiceID],[Transact
ionDate],[AmountExcludingTax],[TaxAmount],[TransactionAmount],
```

```
[OutstandingBalance],[FinalizationDate],[IsFinalized],
[LastEditedBy],[LastEditedWhen])
```

Adding this index definitely increases the performance of your query. It reduces logical reads from 1126 to 312. This is not a tremendous savings, and likely doesn't merit adding in a strict OLTP system, as for every change to the Sales.CustomerTransactions table, all of these column values are copied again to the index pages. For a reporting database, missing indexes can be great things to add, but you always need to take caution.

The Missing Indexes tip is basically where the optimizer was working through what it was looking for to perform the query the fastest, and an index that would have helped was discovered. In Chapter 4, "Optimize database objects and SQL infrastructure," we explore the missing indexes dynamic management view (DMV) where you can see indexes that SQL Server would like to have had for the queries that have been optimized over time. Many of them overlap with other indexes that it has suggested. If you added all of the indexes that it suggested to a busy system, it would be brought to its knees maintaining indexes.

One last property of included columns is important to understand. Included columns in an index can never be used to seek for rows or for ordered scans (since they are not ordered at all), but they can be used to cover a query even if the key columns are not involved. For example, consider the following query that uses the columns that we indexed in the index named ContactPersonID_Include_OrderDate_ExpectedDeliveryDate. If we only reference the OrderDate and ExpectedDeliveryDate in a query, even as a predicate, the index can be scanned instead of the (typically) much larger data in the base table. Take the following query:

Click here to view code image

```
SELECT OrderDate, ExpectedDeliveryDate
FROM  Sales.Orders
WHERE OrderDate > '2015-01-01';
```

Figure 1-17 shows that it uses the index with included columns:

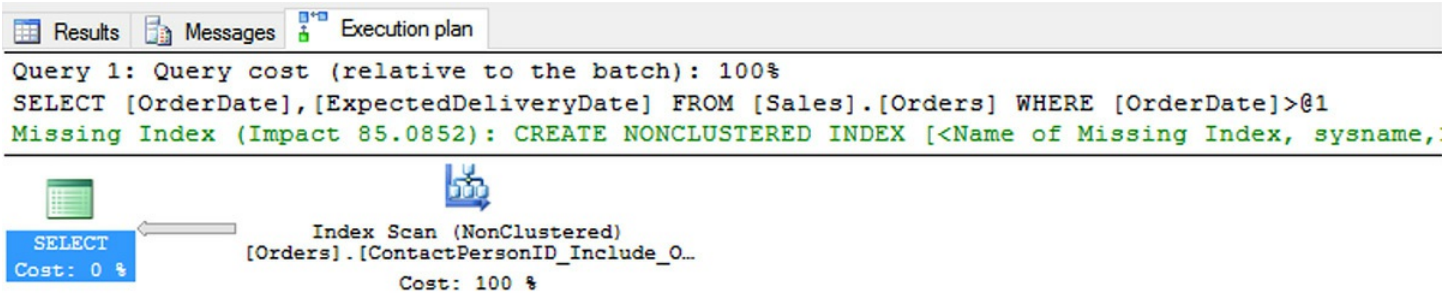


FIGURE 1-17 Query plan showing an index scan of the included columns of the index

Of course, this is not the optimum index for the query, so the query plan suggests the following index, which orders the data on the OrderDate, and includes the ExpectedDeliveryDate as an included column:

Click here to view code image

```
CREATE NONCLUSTERED INDEX [<Name of Missing Index, sysname,>]
ON [Sales].[Orders] ([OrderDate])
INCLUDE ([ExpectedDeliveryDate]);
```

Take caution when using the missing index hints (or missing indexes DMV that are reviewed in Chapter 4.) They are not the best for your system's overall performance, which certainly could be a topic on the exam. Still, covering queries using the INCLUDE feature is a great way to improve situations where a read-intensive workload is using scans to resolve queries because of a few columns that could be added to the index leaf nodes.

Implement clustered index columns by using best practices

The choice of the clustered index can be a complex one as you consider the possibilities. As we have seen throughout the chapter, for various reasons the clustered index is the most important index on your objects. The following are a few characteristics that we need to consider when choosing the clustered index:

- The clustered index contains all of the data of the table on the leaf index pages (or at least the base structures, as data can overflow onto multiple pages for very large rows), so when the clustered index optimally satisfies the query's needs, the performance of the query is going to be better than otherwise.
- The *clustering key* (the term used for the key column(s) of the clustered index) affects the other row store indexes in the table. Picking a larger clustering key could be terrible for performance of all of the other indexes since every non-clustered index key carries this value around.
- If the clustered index is not created as unique, when a duplicate key value is created, it has a four-byte *uniqueifier* attached to duplicated values to make it unique so it makes a proper row locator.
- If you change the value of the clustering key, you change the data on every non-clustered rowstore index.
- The best clustering key is an increasing value, as it is inserted at the end of the structure that is sorted, leading to minimized page splits. When a new row is created in the middle of the clustering key sequence, a large row causes the page split.
- There is also a clustered columnstore index that we cover in Skill 1.4.

Need More Review? More information about Clustered Indexes

An excellent, if slightly older, resource to study more about clustered indexes is Kimberly Tripp's blog series here: *http://www.sqlskills.com/blogs/kimberly/category/clustering-key/*.

So with all of these limitations, what should you choose for the clustered index? There are a few key scenarios to look for:

- **The column(s) that are used for single row fetches, often for modifications** Many OLTP databases operate by fetching a set of rows to a client, and then updating the rows, one at a time. Watching the patterns of usage can help, but almost all of the time the primary key fits this usage, whether the database is using a natural key as primary key, or an artificial surrogate key. An IDENTITY or SEQUENCE based artificial key meets all of the points we started out with, so when implementing a system with artificial keys, it is often the best choice.
- **Range queries** Having all the data in a particular order can be essential to performance when the data that you often need to get a range. Even a range of 1 distinct value makes sense, for a situation like a child row, such as Invoice and InvoiceLineItem where you are constantly fetching InvoiceLineItem rows by the InvoiceId for the invoice.
- **Queries that return large result sets** If you have a situation where a particular query (or set of queries) is run frequently and returns a lot of rows, performing these searches by clustered index can be beneficial.

The typical default that most designers use is to use the clustered index on columns of the primary key. It is always unique, and is almost certainly where the largest percentage of rows are fetched or the PRIMARY KEY constraint is likely misused. In a real database, this requires testing to see how it affects overall performance.

Instinctively, it seems that you want to use the value that the user does most searches on, but the reason that the index that backs the PRIMARY KEY constraint is chosen is because beyond searches, you see lots of fetches by the primary key since singleton SELECTs, UPDATEs, and DELETEs all typically use the primary key for access. Add to that how JOIN operations are done using the primary key, and there needs to be a very compelling case to use something other than the primary key.

Exam Tip

On the exam, you should think about the exact usage of indexes that is being described to make sure what particular need is being emphasized. Questions won't be tricky, but they don't telegraph the answer, so you need to understand the structure of indexes and the usage that the question writer has in mind.

What data type you choose for the clustering key is a matter of opinion and tooling. For the exam, it is good to understand the different possibilities and some of the characteristics of each, particularly when creating an artificial value for a key. Two choices stand out as very common approaches:

- Integer Data types
- GUIDs

It isn't impossible to use different approaches to these, but these are very common in almost any database. In the WideWorldImporters database, all of the primary keys are based on integers. Integers are generally the favored method because they are very small, and are easy to create in a monotonically increasing sequence using the IDENTITY property on a column, or using a SEQUENCE object applied using a DEFAULT constraint.

While integer-based data types generally fit the pattern of a great clustering key, there is another possibility that is very common. Using the uniqueidentifier data type, you can store a standard GUID (Globally Unique Identifier). A major advantage of these values are that they can be generated outside of the database server by any client, unlike an integer value, due to concurrency concerns. However, a major downside is indexing them. They have a 16-byte binary value with a 36-character representation (which can be needed if you have a client that can't handle a GUID), and they are random in nature in terms of sorting. This leads to data being spread around the indexing structures, causing fragmentation, which can reduce the system's ability to scan the data (though this is a bit less of a concern when you have fast SSD drives). You can generate GUID values in the database using NEWID(), or, if you almost never have new values coming from the client, you can use NEWSEQUENTIALID() to generate GUID values that are always increasing, making it a slightly better clustering key than a normal GUID. (However, NEWSEQUENTIALID() even can't be trusted completely because the sequence of GUIDs is not guaranteed to be sequential with other GUIDs created after a reboot.)

In the end, the question of clustering key is very much centered on performance of your queries. Using a natural key can be difficult due to the size of many natural keys, but at the same time, a lot matters about how the application works, and how it turns out that data is used.

Need More Review? The CREATE INDEX Statement

Indexes are a complex topic, and there are a lot of settings that we do not touch on or even mention. It would be very good to review the many settings of the CREATE INDEX statement here in the MSDN library: *https://msdn.microsoft.com/en-us/library/ms188783.aspx*.

Recommend new indexes based on query plans

In the preceding sections on indexing, we used query plans to show that an index made a difference in the performance of one or more queries. The process of reviewing a query plan to determine what the optimizer is doing, or planning to do to optimize a query, is an important one. In this section, we review some of the factors you need to look for in a query plan.

In the code shown in Listing 1-1, we make a copy of a couple of tables from the WideWorldImporters database, with limited indexes to serve as an example.

LISTING 1-1 Setting up a scenario for demonstrating query plans and indexes

Click here to view code image

```
--2074 Rows
SELECT *
INTO   Examples.PurchaseOrders
FROM   WideWorldImporters.Purchasing.PurchaseOrders;

ALTER  TABLE Examples.PurchaseOrders
    ADD CONSTRAINT PKPurchaseOrders PRIMARY KEY
(PurchaseOrderId);

--8367 Rows
SELECT *
INTO   Examples.PurchaseOrderLines
FROM   WideWorldImporters.Purchasing.PurchaseOrderLines;

ALTER  TABLE Examples.PurchaseOrderLines
    ADD CONSTRAINT PKPurchaseOrderLines PRIMARY KEY
(PurchaseOrderLineID);

ALTER TABLE Examples.PurchaseOrderLines
    ADD CONSTRAINT
FKPurchaseOrderLines_Ref_Examples_PurchaseOrderLines
        FOREIGN KEY (PurchaseOrderId) REFERENCES
                                    Examples.PurchaseOrders(P
```

Then we execute the following two queries:

Click here to view code image

```
SELECT *
FROM   Examples.PurchaseOrders
WHERE  PurchaseOrders.OrderDate BETWEEN '2016-03-10' AND '2016-
03-14';

SELECT PurchaseOrderId, ExpectedDeliveryDate
FROM   Examples.PurchaseOrders
WHERE  EXISTS (SELECT *
                 FROM  Examples.PurchaseOrderLines
                 WHERE PurchaseOrderLines.PurchaseOrderId =
                                            PurchaseOrders.P
  AND  PurchaseOrders.OrderDate BETWEEN '2016-03-10' AND '2016-
```

```
03-14' ;
```

Executing these queries returns two sets of 5 rows each, and will probably take much less than a second on any computer as there are not very many rows in these tables at all. Since the query executes so quickly, the developer may get the idea that the query's performance is optimum, even though there will be many users, and much more data in the production version. Using the Query; Display Estimated Query Plan menu in SQL Server Management Studio, we view the estimated plan in Figure 1-18 for this query, to help determine if the query is optimum.

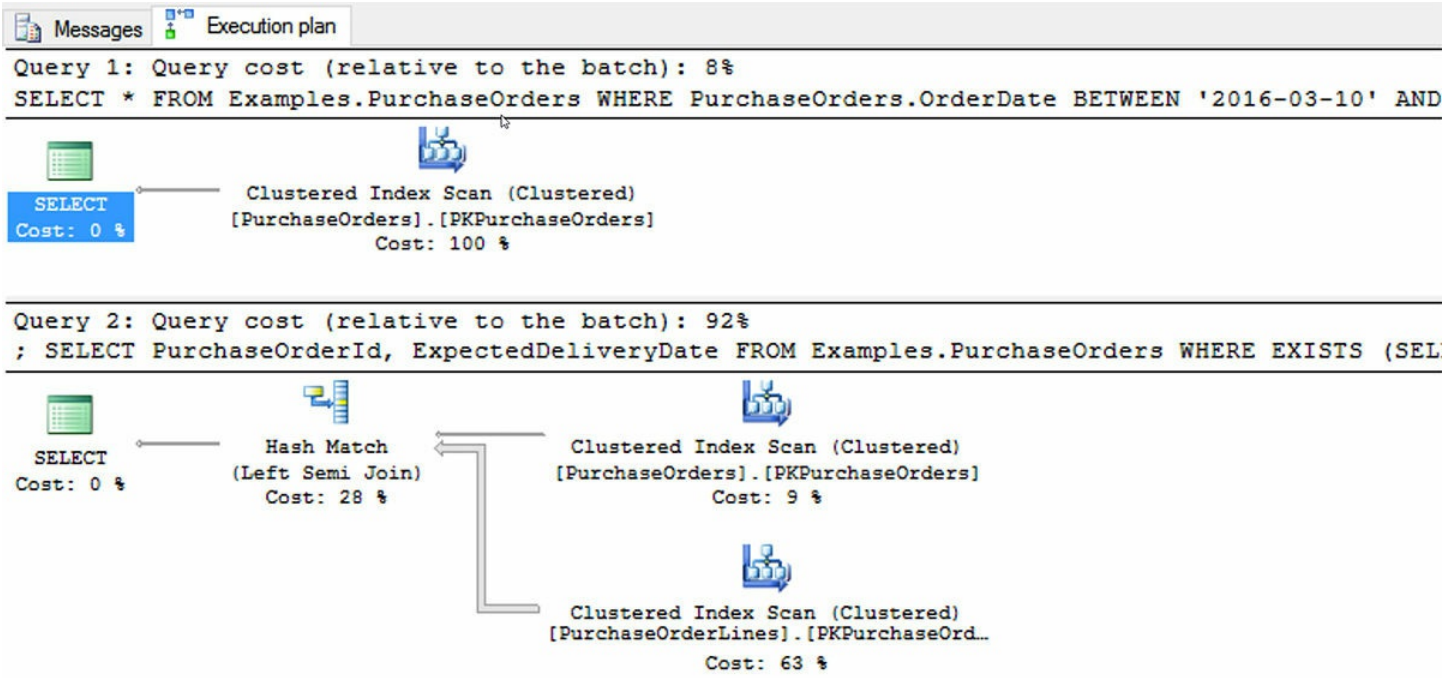


FIGURE 1-18 Query plan for the untuned query on the pair of tables

> **Note An estimated plan**
>
> Earlier in the "Indexing once data is in your tables" section, we covered the multiple ways to see the query plan for a query. For this section we simply look at the estimated plan for the queries.

Even though the first query would have appeared to be the costliest from looking at the results (or at least the equivalent amount of cost, it returned all of the data in the table, while the other just two small columns, all from the same rows), we can see that the first query was considerably less costly. Note how there are hidden costs, such as using user-defined functions that may not show up in a query plan.

For an exam question, you might be asked what you can tell about the indexes on the table from the plan given. We can tell that both queries are scanning the entire physical structures they reference due to the Clustered Index Scan operators. This certainly means that no index is available on the OrderDate column of the Examples.PurchaseOrders table

that can help make this query execute faster. Knowing the row counts, there must be statistics on the OrderDate column that tells the optimizer how many rows will likely match the predicate because the line from PKPurchaseOrders is much smaller than that from PKPurchaseOrderLines. You can see the row counts by hovering over the lines, as shown in Figure 1-19, which is a composite from both lines. (Statistics, their meaning, and how they can be incorrect at times is covered in Skill 4.1).

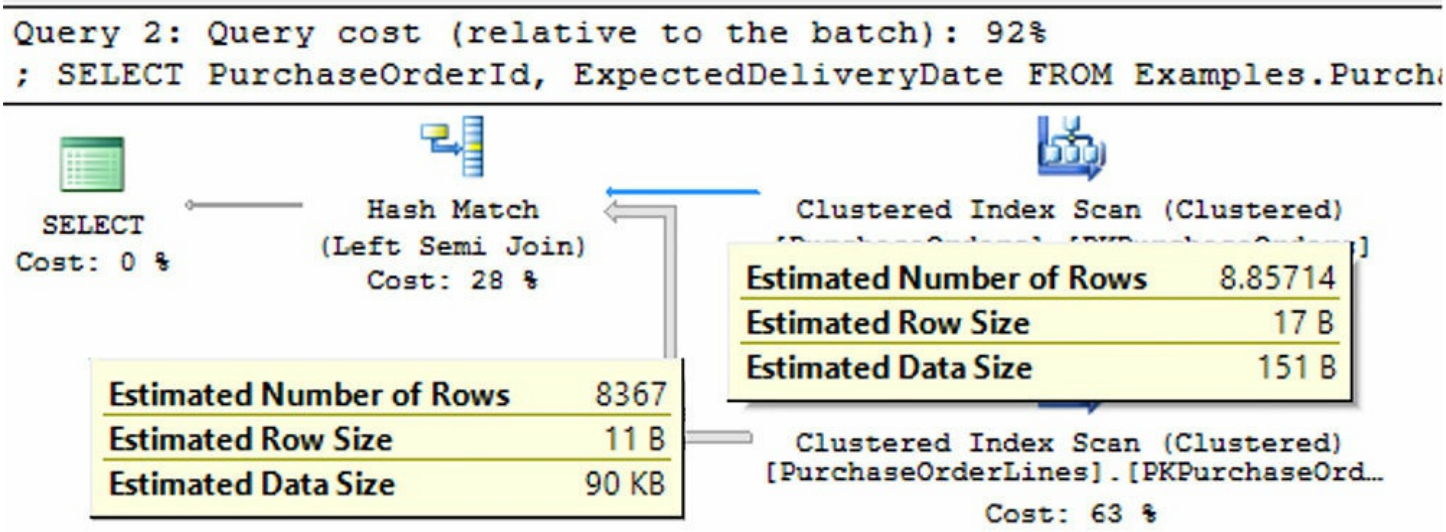


FIGURE 1-19 Query plan for the untuned pair of tables showing number of rows for each operator

Even though it was estimated that approximately 9 rows met the criteria, the optimizer still chose to scan the Examples.PurchaseOrderLines table and use a Hash Match join operator. This is an indication that there is no index on the PurchaseOrderLines.PurchaseOrderId column. Notice that the Hash Match operator is a Left Semi Join. A semi join means that data is returned from the left input, and not the right. This tells you that the Hash Match operator is most likely implementing a filter via a subquery, and not a JOIN in the FROM clause of the query.

Another question you might be asked is what index or indexes are useful to optimize a query based on the plan. From the evidence presented in our example, we can possibly add two indexes. One of them is simple:

Click here to view code image

```
CREATE INDEX PurchaseOrderId ON Examples.PurchaseOrderLines
(PurchaseOrderId);
```

This index is definite since the only data that is used in the Examples.PurchaseOrderLines table is the PurchaseOrderId. What is more complex is whether the following index would be valuable on OrderDate in Examples.PurchaseOrders.:

Click here to view code image

```
CREATE INDEX OrderDate ON Examples.PurchaseOrders (OrderDate);
```

Because of the bookmark lookup factor, and the smaller table, this index is not used for this query, and even more importantly for taking the exam. Without executing the query we could not accurately predict this. However, a better index that will be useful on any size table will include the ExpectedDeliveryDate column along with the OrderDate:

Click here to view code image

```
CREATE INDEX OrderDate_Incl_ExpectedDeliveryDate
     ON Examples.PurchaseOrders (OrderDate) INCLUDE
(ExpectedDeliveryDate);
```

This is because it covers all of the data needed to answer the conditions of the query. On the exam, it will be important to watch for conditions like this, where you are looking for the best answer that is always correct.

Reading a query plan is an essential developer skill, and should be expected on the exam since it is called out specifically in the indexes skill, and skill 4.2 is entitled "Analyze and troubleshoot query plans." All throughout this chapter you will find query plans to demonstrate multiple scenarios you can see from the query plan. It is the primary way that we can tell how well a query is tuned.

Note Query plans

For deep detail on query plans, one of the best resources is "SQL Server Execution Plans, Second Edition" by Grant Fritchey: *https://www.simple-talk.com/books/sql-books/sql-server-execution-plans,-second-edition,-by-grant-fritchey/*.

Skill 1.3: Design and implement views

A view is a single SELECT statement that is compiled into a reusable object. Views can be used in a variety of situations, for a variety of purposes. To the user, views more or less appear the same as tables, and have the same security characteristics as tables. They are objects just like tables (and procedure, sequences, etc.) and as such cannot have the same name as any other object. The query can be very complex, or very simple. Just like a table, it does not have parameters. Also like a table, there is no guaranteed order of the data in a view, even if you have an ORDER BY clause in the view to support the TOP phrase on the SELECT clause.

The basic form of a view is very straightforward:

Click here to view code image

```
CREATE VIEW SchemaName.ViewName
[WITH OPTIONS]
AS SELECT statement
[WITH CHECK OPTION]
```

The SELECT statement can be as complex as you desire, and can use CTEs, set operators like UNION and EXCEPT, and any other constructs you can use in a single statement.

The options you can specify are:

- **SCHEMABINDING** Protects the view from changes to the objects used in the SELECT statement. For example, if you reference Table1.Column1, the properties of that Column1 cannot be changed, nor can Table1 be dropped. Columns, not references can be removed, or new columns added.
- **VIEW_METADATA** Alters how an application that accesses the VIEW sees the metadata. Typically, the metadata is based on the base tables, but VIEW_METADATA returns the definition from the VIEW object. This can be useful when trying to use a view like a table in an application.
- **ENCRYPTION** Encrypts the entry in sys.syscomments that contains the text of the VIEW create statement. Has the side effect of preventing the view from being published as part of replication.

THE WITH CHECK OPTION will be covered in more detail later in this skill, but it basically limits what can be modified in the VIEW to what could be returned by the VIEW object.

This section covers how to:

- Design a view structure to select data based on user or business requirements

- Identify the steps necessary to design an updateable view
- Implement partitioned views
- Implement indexed views

Design a view structure to select data based on user or business requirements

There are a variety of reasons for using a view to meet user requirements, though some reasons have changed in SQL Server 2016 with the new Row-Level Security feature (such as hiding data from a user, which is better done using the Row-Level Security feature, which is not discussed in this book as it is not part of the objectives of the exam.)

For the most part, views are used for one specific need: to query simplification to encapsulate some query, or part of query, into a reusable structure. As long as they are not layered too deeply and complexly, this is a great usage. The following are a few specific scenarios to consider views for:

- **Hiding data for a particular purpose** A view can be used to present a projection of the data in a table that limits the rows that can be seen with a WHERE clause, or by only returning certain columns from a table (or both).
- **Reformatting data** In some cases, there is data in the source system that is used frequently, but doesn't match the need as it stands. Instead of dealing with this situation in every usage, a view can provide an object that looks like the customer needs.
- **Reporting** Often to encapsulate a complex query that needs to perform occasionally, and even some queries that aren't exactly complicated, but they are performed repeatedly. This can be for use with a reporting tool, or simply for ad-hoc usage.
- **Providing a table-like interface for an application that can only use tables** Sometimes a stored procedure makes more sense, but views are a lot more general purpose than are stored procedures. Almost any tool that can ingest and work with tables can use a view.

Let's examine these scenarios for using a VIEW object, except the last one. That particular utilization is shown in the next section on updatable views. Of course, all of these scenarios can sometimes be implemented in the very same VIEW, as you might just want to see sales for the current year, with backordered and not shipped products grouped together for a report, and you might even want to be able to edit some of the data using that view. While this might be possible, we look at them as individual examples in the following sections.

Using views to hide data for a particular purpose

One use for views is to provide access to certain data stored in a table (or multiple tables).

For example, say you have a customer requirement that states: “We need to be able to provide access to orders made in the last 12 months (to the day), where there were more than one-line items in that order. They only need to see the Line items, Customer, SalesPerson, Date of Order, and when it was likely to be delivered by.”

A view might be created as shown in Listing 1-2.

LISTING 1-2 Creating a view that meets the user requirements

Click here to view code image

```
CREATE VIEW Sales.Orders12MonthsMultipleItems
AS
SELECT OrderId, CustomerID, SalespersonPersonID, OrderDate,
ExpectedDeliveryDate
FROM   Sales.Orders
WHERE  OrderDate >= DATEADD(Month,-12,SYSDATETIME())
  AND (SELECT COUNT(*)
       FROM   Sales.OrderLines
       WHERE  OrderLines.OrderID = Orders.OrderID) > 1;
```

Now the user can simply query the data using this view, just like a table:

Click here to view code image

```
SELECT TOP 5 *
FROM   Sales.Orders12MonthsMultipleItems
ORDER BY ExpectedDeliveryDate desc;
```

Using TOP this returns 5 rows from the table:

Click here to view code image

```
OrderId     CustomerID  SalespersonPersonID
OrderDate  ExpectedDeliveryDate
----------- ----------- ------------------- ---------- --------
------------
73550       967         15                  2016-05-31 2016-06-
01
73549       856         16                  2016-05-31 2016-06-
01
73548       840         3                   2016-05-31 2016-06-
01
73547       6           14                  2016-05-31 2016-06-
01
73546       810         3                   2016-05-31 2016-06-
01
```

Note that this particular usage of views is not limited to security like using row-level

security might be. A user who has access to all of the rows in the table can still have a perfectly valid reason to see a specific type of data for a purpose.

Using a view to reformatting data in the output

Database designers are an interesting bunch. They often try to store data in the best possible format for space and some forms of internal performance that can be gotten away with. Consider this subsection of the Application.People table in WideWorldImporters database.

Click here to view code image

```
SELECT PersonId, IsPermittedToLogon, IsEmployee, IsSalesPerson
FROM   Application.People;
```

What you see is 1111 rows of kind of cryptic data to look at (showing the first four rows):

Click here to view code image

```
PersonId    IsPermittedToLogon IsEmployee IsSalesPerson
----------- ------------------ ---------- -------------
1           0                  0          0
2           1                  1          1
3           1                  1          1
4           1                  1          0
```

A common request from a user that needs to look at this data using Transact-SQL could be: "I would like to see the data in the People table in a more user friendly manner. If the user can logon to the system, have a textual value that says 'Can Logon', or 'Can't Logon' otherwise. I would like to see employees typed as 'SalesPerson' if they are, then as 'Regular' if they are an employee, or 'Not Employee' if they are not an employee."

In Listing 1-3 is a VIEW object that meets these requirements.

LISTING 1-3 Creating the view reformat some columns in the Application.People table

Click here to view code image

```
CREATE VIEW Application.PeopleEmployeeStatus
AS
SELECT PersonId, FullName,
       IsPermittedToLogon, IsEmployee, IsSalesPerson,
       CASE WHEN IsPermittedToLogon = 1 THEN 'Can Logon'
             ELSE 'Can''t Logon' END AS LogonRights,
       CASE WHEN IsEmployee = 1 and IsSalesPerson = 1
                THEN 'Sales Person'
            WHEN IsEmployee = 1
```

```
                    THEN 'Regular'
                ELSE 'Not Employee' END AS EmployeeType
FROM   Application.People;
```

Now, querying the data in the same manner (leaving off names), you see something more pleasant to work with:

Click here to view code image

```
SELECT PersonId, LogonRights, EmployeeType
FROM   Application.PeopleEmployeeStatus;
```

Which returns:

Click here to view code image

```
PersonId    LogonRights EmployeeType
----------- ----------- -------------
1           Can't Logon Not Employee
2           Can Logon   Sales Person
3           Can Logon   Sales Person
4           Can Logon   Regular
```

There is one serious downside to this method of reformatting. While this looks better, and is easier to see, queries that use the reformatted values to filter on the new columns never use an index since the data does not match what is in the index. For a smaller table, this isn't an issue, but it is a concern. We included the columns in the view that had the original data for that reason.

A final concern is not to use views as layers of encapsulation in your application code for that same reason. The more layers of views you have, the less likely you get a great plan of execution. Views are definitely useful to have for morphing a set for many reasons, particularly when a user is repeatedly doing the same kinds of transformations in their code.

Using a view to provide a reporting interface

A very useful pattern to apply with a view is building a reporting interface, to format some data for a reporting tool.

Requirements might be given to "Build a simple reporting interface that allows us to see sales profit or net income broken down by city, state, or territory customer category for the current week, up to the most current data". If the system is normalized, there are quite a few tables involved in the query. Note that an important part of these requirements is that it be up to the most current data. If it did not include the most recent data, a data warehousing solution with a separate database would likely be more efficient.

In Listing 1-4, the code for the view that gives you a structure that can easily be used for providing these answers is included. The object is in a new schema named Reports to

segregate it from other bits of code, and the view is suffixed "Basis" because this view could be the basis of several reports.

There are not any locking or isolation hints, and it is generally not a good practice to do so in your code unless using the SNAPSHOT isolation level in your database. Chapter 3 covers concurrency, isolation levels, in more detail.

LISTING 1-4 Creating the view that is the basis of an Invoice Summary report

Click here to view code image

```
CREATE SCHEMA Reports;
GO
CREATE VIEW Reports.InvoiceSummaryBasis
AS
SELECT Invoices.InvoiceId,
CustomerCategories.CustomerCategoryName,
       Cities.CityName, StateProvinces.StateProvinceName,
       StateProvinces.SalesTerritory,
       Invoices.InvoiceDate,
       --the grain of the report is at the invoice, so total
       --the amounts for invoice
       SUM(InvoiceLines.LineProfit) as InvoiceProfit,
       SUM(InvoiceLines.ExtendedPrice) as InvoiceExtendedPrice
FROM   Sales.Invoices
         JOIN Sales.InvoiceLines
            ON Invoices.InvoiceID = InvoiceLines.InvoiceID
         JOIN Sales.Customers
              ON Customers.CustomerID = Invoices.CustomerID
         JOIN Sales.CustomerCategories
              ON Customers.CustomerCategoryID =
                                 CustomerCategories.CustomerCateg
         JOIN Application.Cities
              ON Customers.DeliveryCityID = Cities.CityID
         JOIN Application.StateProvinces
              ON StateProvinces.StateProvinceID =
Cities.StateProvinceID
GROUP BY Invoices.InvoiceId,
CustomerCategories.CustomerCategoryName,
       Cities.CityName, StateProvinces.StateProvinceName,
       StateProvinces.SalesTerritory,
       Invoices.InvoiceDate;
```

Now you can create a report of the top 5 Sales by SalesTerritory pretty simply:

Click here to view code image

```
SELECT TOP 5 SalesTerritory, SUM(InvoiceProfit) AS
InvoiceProfitTotal
FROM Reports.InvoiceSummaryBasis
WHERE InvoiceDate > '2016-05-01'
GROUP BY SalesTerritory
ORDER BY InvoiceProfitTotal DESC;
```

This returns:

Click here to view code image

```
SalesTerritory           InvoiceProfitTotal
------------------------ ---------------------
Southeast                536367.60
Great Lakes              366182.65
Mideast                  344703.00
Southwest                344386.95
Plains                   288766.20
```

Or, using the same structure, the top five sales by state and customer category:

Click here to view code image

```
SELECT TOP 5 StateProvinceName, CustomerCategoryName,
       SUM(InvoiceExtendedPrice) AS InvoiceExtendedPriceTotal
FROM Reports.InvoiceSummaryBasis
WHERE InvoiceDate > '2016-05-01'
GROUP BY StateProvinceName, CustomerCategoryName
ORDER BY InvoiceExtendedPriceTotal DESC;
```

This returns:

Click here to view code image

```
StateProvinceName    CustomerCategoryName   InvoiceExtendedPric
-------------------- ---------------------- -------------------
--------------------
Texas                Novelty Shop           229966.31
Pennsylvania         Novelty Shop           210254.62
Ohio                 Novelty Shop           201242.59
New York             Novelty Shop           197664.32
California           Novelty Shop           178698.48
```

Clearly, novelty shops are a big business for WideWorldImporters. Looking at the plans of both queries, you should notice a couple of things. The plans are pretty large, not surprising since we joined a lot of tables. Nothing stands out as too terrible, and both plans suggest the same index:

Click here to view code image

```
CREATE NONCLUSTERED INDEX [<Name of Missing Index, sysname,>]
```

```
        ON [Sales].[Invoices] ([InvoiceDate]) INCLUDE
([InvoiceID],[CustomerID]);
```

There is no need to actually apply the index, but as we have discussed frequently, having a covering index for almost every query is a very common desire of the optimizer, and is not always a bad idea when you are doing ranges queries on the table (which was one of the strengths of the clustered index, because it had all of the data sorted for the range. Covering the data needs of queries make a non-clustered index with included columns behave like a clustered index for queries such as this.)

Identify the steps necessary to design an updateable view

In the previous section, we identified four scenarios where views are frequently useful (not an exhaustive list). The fourth scenario (providing a table-like interface for an application that can only use tables) was put off until this section because for the most part, the goal is to make objects that behave exactly like tables in regards to SELECT, INSERT, UPDATE, and DELETE statements with no special modifications.

To provide interfaces with views there are some special configurations that you need to know. We look at the following scenarios for using views to modify tables.

- Modifying views that reference one table
- Limiting what data can be added to a table through a view through DDL
- Modifying data in views with more than one table

Modifying views that reference one table

Generally speaking, any view that references a single table is going to be editable. For example, create a VIEW on the HumanResources.Employee table. To keep it simple, say the requirements are: “The user needs a view to allow the interface to only be able to modify rows where the type of gadget is ‘Electronic’, but not any other value.” The table is shown in Listing 1-5.

LISTING 1-5 Creating the table and some data that is the basis of the updatable view example

Click here to view code image

```
CREATE TABLE Examples.Gadget
(
    GadgetId    int NOT NULL CONSTRAINT PKGadget PRIMARY KEY,
    GadgetNumber char(8) NOT NULL CONSTRAINT AKGadget UNIQUE,
    GadgetType  varchar(10) NOT NULL
);
INSERT INTO Examples.Gadget(GadgetId, GadgetNumber, GadgetType)
VALUES  (1,'00000001','Electronic'),
```

```
        (2,'00000002','Manual'),
        (3,'00000003','Manual');
```

When building a view to be editable, the simpler the view, the easier it is when you're working with it. In the code that you see in Listing 1-6, there is a column that is the uppercase version of the gadget type to show how a non-executable column behaves.

LISTING 1-6 Creating the view that is the basis of an Invoice Summary report

Click here to view code image

```
CREATE VIEW Examples.ElectronicGadget
AS
    SELECT GadgetId, GadgetNumber, GadgetType,
           UPPER(GadgetType) AS UpperGadgedType
    FROM   Examples.Gadget
    WHERE GadgetType = 'Electronic';
```

Note Using VIEW_METADATA

When using views to provide an interface for an application, you can use the VIEW_METADATA to alter how an application that accesses the view sees the metadata. Typically, the metadata is based on the base tables, but VIEW_METADATA returns the definition from the VIEW object. This can be useful when trying to use a view like a table in an application.

Now, any user who is granted access to this view can only see rows that meet the WHERE clause of GadgetType = 'Electronic'.

Exam Tip

It's very useful to know all of the comparison operators that are used in SQL, such as LIKE, IN, =, <,>,<> (or !=), etc. While there aren't necessarily questions on how to use LIKE, base knowledge of logical and comparison operators is necessary to get a question correct.

Next, perform the following query that shows you the data in the table, seeing what data the user sees:

Click here to view code image

```
SELECT ElectronicGadget.GadgetNumber AS FromView,
Gadget.GadgetNumber AS FromTable,
```

```
        Gadget.GadgetType, ElectronicGadget.UpperGadgetType
FROM   Examples.ElectronicGadget
         FULL OUTER JOIN Examples.Gadget
            ON ElectronicGadget.GadgetId = Gadget.GadgetId;
```

You can see that for rows where the GadgetType <> 'Electronic':

Click here to view code image

```
FromView FromTable GadgetType UpperGadgetType
-------- --------- ---------- ---------------
00000001 00000001  Electronic ELECTRONIC
NULL     00000002  Manual     NULL
NULL     00000003  Manual     NULL
```

Now we run three statements to create some new rows, delete two rows, and update two rows in the table. In the comments on the code, include details of what you're doing. First, try creating two new rows, referencing the derived column:

Click here to view code image

```
INSERT INTO Examples.ElectronicGadget(GadgetId, GadgetNumber,
                                      GadgetType,
UpperGadgetType)
VALUES (4,'00000004','Electronic','XXXXXXXXXX'), --row we can
see in view
       (5,'00000005','Manual','YYYYYYYYYY'); --row we cannot
see in view
```

This fails, as you would expect:

Click here to view code image

```
Msg 4406, Level 16, State 1, Line 433
Update or insert of view or function
'Examples.ElectronicGadget' failed because
it contains a derived or constant field.
```

Exam Tip

As you are studying for the exam, it is a great idea to try different syntaxes and scenarios everything to see what happens. While the question writers are not writing tricky questions, they can test you on scenarios that are atypical based on your experience. By pushing the boundaries of what you might normally try, you learn new concepts.

Now, try again, not referencing the calculated column:

Click here to view code image

```
INSERT INTO Examples.ElectronicGadget(GadgetId, GadgetNumber,
GadgetType)
VALUES (4,'00000004','Electronic'),
       (5,'00000005','Manual');
```

This succeeds, so now use the query with the FULL OUTER JOIN from before, but limit it to the rows you created.

Click here to view code image

```
SELECT ElectronicGadget.GadgetNumber as FromView,
Gadget.GadgetNumber as FromTable,
        Gadget.GadgetType, ElectronicGadget.UpperGadgetType
FROM   Examples.ElectronicGadget
         FULL OUTER JOIN Examples.Gadget
            ON ElectronicGadget.GadgetId = Gadget.GadgetId;

WHERE Gadget.GadgetId in (4,5);

Both rows were created, even though you cannot see the row in
the view after the
operation:

FromView FromTable GadgetType UpperGadgetType
-------- --------- ---------- ---------------
00000004 00000004  Electronic ELECTRONIC
NULL     00000005  Manual     NULL
```

Next, update two rows:

Click here to view code image

```
--Update the row we could see to values that could not be seen
UPDATE Examples.ElectronicGadget
SET    GadgetType   = 'Manual'
WHERE  GadgetNumber = '00000004';

--Update the row we could see to values that could actually see
UPDATE Examples.ElectronicGadget
SET    GadgetType   = 'Electronic'
WHERE  GadgetNumber = '00000005';
```

When looking at the data (using the same query as before,) see that the row you could see has change to be not visible from the view, but the row we could not see was not updated:

Click here to view code image

```
FromView FromTable GadgetType UpperGadgetType
-------- --------- ---------- ---------------
```

```
NULL      00000004  Manual      NULL
NULL      00000005  Manual      NULL
```

Since you cannot see the row in the results of a query of the view, you cannot update the row either. Hence, the same would be true of the DELETE operation. The FROM clause of the SELECT, DELETE, and UPDATE all work the same, and only give us access to the rows that are visible through the view. What is interesting though is that you are able to update a row from a name you could see, to a name you could not. The following section demonstrates how to make that not the case using DDL. Change the E1111111 row back to the original value:

Click here to view code image

```
UPDATE Examples.Gadget
SET    GadgetType   = 'Electronic'
WHERE  GadgetNumber = '00000004';
```

Exam Tip

Note that any constraints on tables that views represent are enforced. We have applied PRIMARY KEY, UNIQUE, and FOREIGN KEY constraints as part of the database design skill earlier. In Chapter 2 we look at all of the constraint types in greater detail.

Limiting what data can be added to a table through a view through DDL

When using a view as an interface like we are doing in this section, one of the things that you generally don't want to occur is to have a DML statement affect the view of the data that is not visible to the user of the view, as we saw in the previous section.

In order to stop this from occurring, there is a clause on the creation of the view called WITH CHECK OPTION that checks to make sure that the result of the INSERT or UPDATE statement is still visible to the user of the view. In Listing 1-7, modify the Examples.ElectronicGadget view to include this clause:

LISTING 1-7 Altering the view to use the WITH CHECK OPTION

Click here to view code image

```
ALTER VIEW Examples.ElectronicGadget
AS
    SELECT GadgetId, GadgetNumber, GadgetType,
           UPPER(GadgetType) AS UpperGadgetType
    FROM   Examples.Gadget
    WHERE GadgetType = 'Electronic'
```

```
WITH CHECK OPTION;
```

Exam Tip

To change the definition of a coded object like a view, stored procedure, etc. you can use the ALTER command, with new code for the object. Another method is to drop the object and recreate it. Dropping the object loses all of the related security, and simply executing DROP VIEW viewName; fails if the view does not exist. In SQL Server 2016, a new version of the DROP command was added: DROP [objectType] IF EXISTS drops the object if it exists. For our current view, it would be DROP VIEW IF EXISTS Examples.ElectronicGadget.

Now, when you attempt to create a new row that would not be visible, you get an error. As an example, try the following:

Click here to view code image

```
INSERT INTO Examples.ElectronicGadget(GadgetId, GadgetNumber,
GadgetType)
VALUES (6,'00000006','Manual');
```

This returns the following error now:

Click here to view code image

```
Msg 550, Level 16, State 1, Line 482
The attempted insert or update failed because the target view
either specifies
WITH CHECK OPTION or spans a view that specifies WITH CHECK
OPTION and one or more
rows resulting from the operation did not qualify under the
CHECK OPTION constraint.
```

This UPDATE worked earlier, but it shouldn't because it does not match the view definition and it gives you back the same error message as the previous one.

Click here to view code image

```
UPDATE Examples.ElectronicGadget
SET    GadgetType   = 'Manual'
WHERE  GadgetNumber = '00000004';
```

While this is definitely different than a normal table, it usually makes more sense because the view has carved out a slice of an object's domain, and it is generally illogical that a change to a row in a table should be able to move data out of the table's domain (be it a table or a view, which is commonly referred to as a *virtual table*).

Modifying data in views with more than one table

So far, the view we have worked with only contained one table. In this section we look at how things are affected when you have greater than one table in the view. Listing 1-8 adds a table to the Examples.Gadget table we have been using in this section on modifying data in views, including data, and a foreign key constraint.

LISTING 1-8 Adding a table to go with the Examples.Gadget table to show a view with more than one table

Click here to view code image

```
CREATE TABLE Examples.GadgetType
(
    GadgetType  varchar(10) NOT NULL CONSTRAINT PKGadgetType
PRIMARY KEY,
    Description varchar(200) NOT NULL
)
INSERT INTO Examples.GadgetType(GadgetType, Description)
VALUES ('Manual','No batteries'),
       ('Electronic','Lots of bats');

ALTER TABLE Examples.Gadget
   ADD CONSTRAINT FKGadget$ref$Examples_GadgetType
       FOREIGN KEY (GadgetType) REFERENCES Examples.GadgetType
(GadgetType);
```

In Listing 1-9 is the code for a view that references both of the tables we have created, with no WHERE clause to limit the rows returned. Note that we have duplicated the GadgetType column from both tables to allow access to both columns.

LISTING 1-9 View that references multiple tables

Click here to view code image

```
CREATE VIEW Examples.GadgetExtension
AS
    SELECT Gadget.GadgetId, Gadget.GadgetNumber,
           Gadget.GadgetType, GadgetType.GadgetType As
DomainGadgetType,
           GadgetType.Description as GadgetTypeDescription
    FROM   Examples.Gadget
             JOIN Examples.GadgetType
                ON Gadget.GadgetType = GadgetType.GadgetType;
```

Now try to insert a new gadget and gadget type simultaneously:

Click here to view code image

```
INSERT INTO Examples.GadgetExtension(GadgetId, GadgetNumber,
GadgetType,
                     DomainGadgetType, GadgetTypeDescription)
VALUES(7,'00000007','Acoustic','Acoustic','Sound');
```

This ends with the following message:

Click here to view code image

```
Msg 4405, Level 16, State 1, Line 512
View or function 'Examples.GadgetExtension' is not updatable
because the modification
affects multiple base tables.
```

However, if you know the internals of the view, and know which columns go with which tables, you can break this up into two statements:

Click here to view code image

```
INSERT INTO Examples.GadgetExtension(DomainGadgetType,
GadgetTypeDescription)
VALUES('Acoustic','Sound');

INSERT INTO Examples.GadgetExtension(GadgetId, GadgetNumber,
GadgetType)
VALUES(7,'00000007','Acoustic');
```

See that it works and, looking at the data, see that both rows have been created. Now, to see the UPDATE work, we update the description of one of the types. There are two rows where the GadgetType = 'Electronic'.

Click here to view code image

```
SELECT *
FROM   Examples.Gadget
             JOIN Examples.GadgetType
                ON Gadget.GadgetType = GadgetType.GadgetType
WHERE  Gadget.GadgetType = 'Electronic';
```

Two rows are returned:

Click here to view code image

```
GadgetId    GadgetNumber GadgetType GadgetType Description
----------- ------------ ---------- ---------- --------------
1           00000001     Electronic Electronic Lots of bats
4           00000004     Electronic Electronic Lots of bats
```

Update one of these rows, using the primary key column, and setting the description:

Click here to view code image

```
UPDATE Examples.GadgetExtension
SET   GadgetTypeDescription = 'Uses Batteries'
WHERE GadgetId = 1;
```

Look at the data again and see that both rows have changed:

Click here to view code image

```
GadgetId    GadgetNumber GadgetType GadgetType Description
----------- ------------ ---------- ---------- ---------------
1           00000001     Electronic Electronic Uses Batteries
4           00000004     Electronic Electronic Uses Batteries
```

There is no way to specify that a DELETE statement affects any certain columns, so DELETE from a view that touches multiple tables doesn't work by simply affecting a DELETE against the view.

> **Note The INSTEAD OF TRIGGER object**
>
> In Skill 2.3 we introduce another very important method of making any view modifiable using an INSTEAD OF TRIGGER object that will allow any view to be editable, even if the view doesn't reference any TABLE objects in the database. We defer that until introducing the concept of building a TRIGGER object.

Implement partitioned views

A partitioned view is a view that is based on a query that uses a UNION ALL set operator to treat multiple tables as one. Before the feature of partitioning tables and indexes was created, it was the primary way to give an administrator the ability to manage multiple "slices" of a table as different physical resources.

The feature still exists, both for backward compatibility (since partitioning is the typical best way to implement partitioning since 2005), and to enable a VIEW object to work across multiple independent *federated* SQL Servers. Generally, the place where this feature is still the best practice is a case such as having two or more servers located in different corporate locations. Each location might have a copy of their data, and then a view is created that lets you treat the table as one on the local server.

Our example is located on a single server, but we will point out where you would address and configure the references to an external server. Listing 1-10 creates two tables and loads them from the Sales.Invoices table in the WideWordImporters database to build a local version of a partitioned view.

LISTING 1-10 Tables and data to use as the basis of a Partitioned View

Click here to view code image

```
CREATE TABLE Examples.Invoices_Region1
(
    InvoiceId   int NOT NULL
        CONSTRAINT PKInvoices_Region1 PRIMARY KEY,
        CONSTRAINT CHKInvoices_Region1_PartKey
                        CHECK (InvoiceId BETWEEN 1 and
10000),
    CustomerId  int NOT NULL,
    InvoiceDate date NOT NULL
);
CREATE TABLE Examples.Invoices_Region2
(
    InvoiceId   int NOT NULL
        CONSTRAINT PKInvoices_Region2 PRIMARY KEY,
        CONSTRAINT CHKInvoices_Region2_PartKey
                        CHECK (InvoiceId BETWEEN 10001 and
20000),

           CustomerId  int NOT NULL,
           InvoiceDate date NOT NULL

        );

        INSERT INTO Examples.Invoices_Region1 (InvoiceId,
CustomerId, InvoiceDate)
        SELECT InvoiceId, CustomerId, InvoiceDate
        FROM   WideWorldImporters.Sales.Invoices
        WHERE  InvoiceId BETWEEN 1 and 10000;

        INSERT INTO Examples.Invoices_Region2 (InvoiceId,
CustomerId, InvoiceDate)
        SELECT InvoiceId, CustomerId, InvoiceDate
        FROM   WideWorldImporters.Sales.Invoices
        WHERE  InvoiceId BETWEEN 10001 and 20000;
```

The PRIMARY KEY constraint of this table must be involved in the partitioning for this to work. In our case, we use a range of InvoiceId values, which is the primary key of both tables. You could use a SEQUENCE object with a pre-defined range to create your data, but the partitioning column cannot be a column with the IDENTITY property, and it cannot be loaded from a DEFAULT constraint. The partitioning range must be enforced with a CHECK constraint, and must be for a mutually-exclusive range of values.

There are several criteria beyond what you have seen thus far that you should understand, but note that not every source table needs the exact same shape, even if that is the more typical application.

> **Need More Review? Configuring Partitioned Views**
>
> There is a complete list and more complete reading on the subject on the MSDN site here: *https://msdn.microsoft.com/en-us/library/ms187956.aspx*. There are several rules that make configuring a partitioned a complex operation that cannot be done with any set of similarly configured tables.

In Listing 1-11, we create a very simple, locally-partitioned view.

LISTING 1-11 Partitioned View created from the tables in Listing 1-10

Click here to view code image

```
CREATE VIEW Examples.InvoicesPartitioned
AS
    SELECT InvoiceId, CustomerId, InvoiceDate
    FROM   Examples.Invoices_Region1
    UNION ALL
    SELECT InvoiceId, CustomerId, InvoiceDate
    FROM   Examples.Invoices_Region2;
```

Using this VIEW object, and requesting data from only one of the TABLE objects by partitioning key only needs to fetch data from one of the partitions. As an example, fetch the row where InvoiceId = 1:

Click here to view code image

```
SELECT *
FROM  Examples.InvoicesPartitioned
WHERE InvoiceId = 1;
```

See the following plan in figure 1-20 that only references the one index.

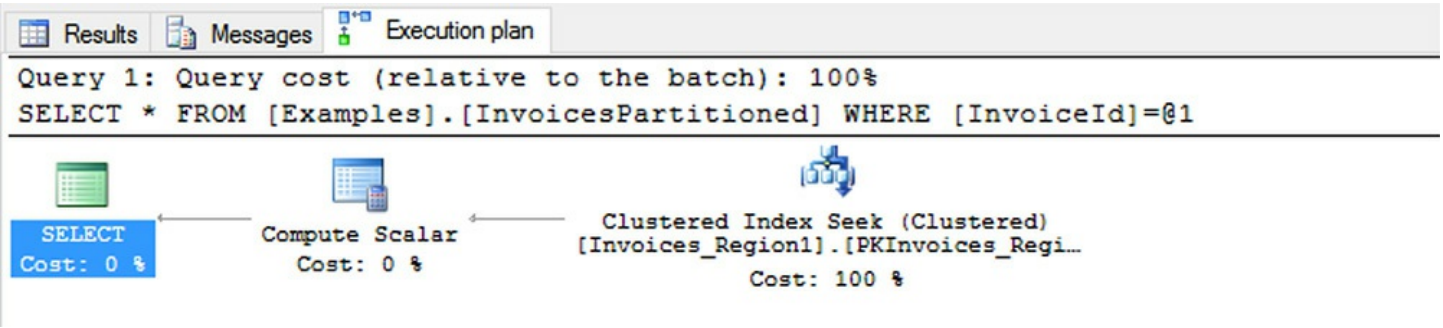


FIGURE 1-20 Plan from query that accesses data in one partition.

Even if you made the query access all 10000 rows in the Sales.Invoices_Region1 table by making the WHERE predicate BETWEEN 1 AND 10000, it would only access the one table. A predicate of IN (1,10001) however, accesses both tables.

More difficult for the user is that all queries need to specify the partitioning key(s) values(s) or the partitioning does not work. For the following query, use the InvoiceDate in the predicate:

Click here to view code image

```
SELECT InvoiceId
FROM   Examples.InvoicesPartitioned
WHERE  InvoiceDate = '2013-01-01';
```

The range of values in the results shows that only include data from the Sales.Invoices_Region1 table is returned.

```
InvoiceId
-----------
1
2
...
40
41
```

Looking at the plan shows the query accessed both physical tables, as seen in Figure 1-21. The Concatenation operator represents the UNION ALL in the query, as it is concatenating the two sets together. You can see from the size of the lines that the optimizer was expecting very few rows, but it still had to perform the scan.

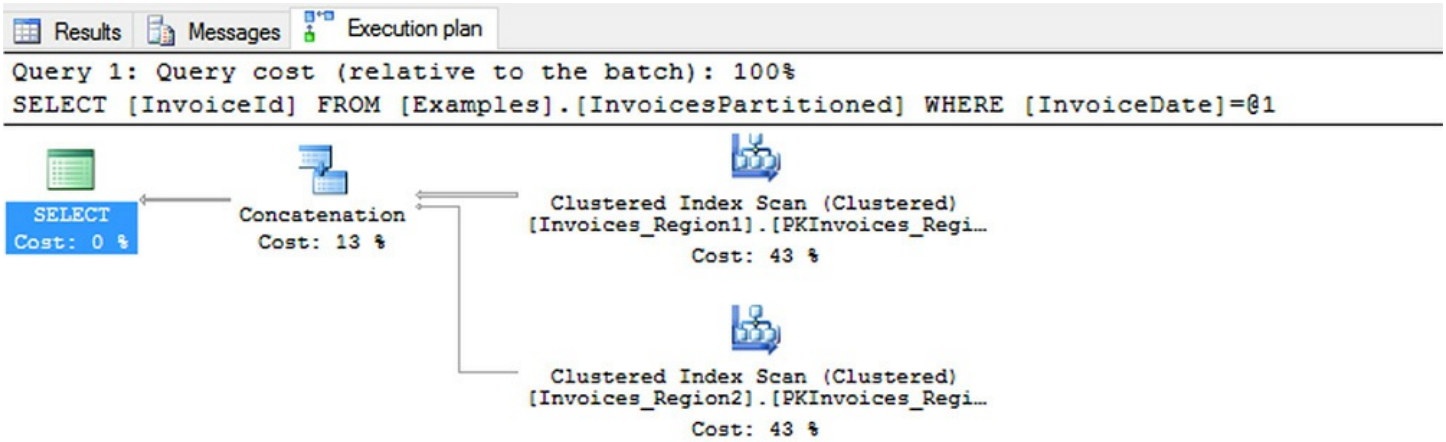


FIGURE 1-21 Plan from query that accesses data in both partitions, yet only returns data from one

When doing this on a local table, much like in a partitioned table, this is generally not so terrible. You could get some performance benefit by locating the tables being in different physical structures, or even different file groups. But what if this was on a different server? If the query that the VIEW object was based upon was changed:

Click here to view code image

```
SELECT InvoiceId, CustomerId, InvoiceDate
FROM   Sales.Invoices_Region1
UNION ALL
SELECT InvoiceId, CustomerId, InvoiceDate
FROM   ServerName.DatabaseName.Sales.Invoices_Region2;
```

And now that the database is on a *linked server*, this is more costly than local and could be very costly to execute the query. A linked server is a server that is registered in SQL Server's metadata with security to access data that is not located on the local server. An indication that a query is using a linked server is when you see the object referenced by a four-part name.

> **Need More Review? Linked servers**
>
> Read more about linked servers here on MSDN: *https://msdn.microsoft.com/en-us/library/ms188279.aspx*)

Implement indexed views

An indexed view (sometimes referred to as a *materialized* view), is a view that has been made into more than just a simple stored query by creating a clustered index on it. By doing this, it basically makes it into a copy of data in a physical structure much like a table.

The first benefit of using an indexed view is that when you use it Enterprise Edition of SQL Server, it uses the stored data in the index structure. For Standard Edition, it uses the code of the query unless you use a NOEXPAND table hint, in which case it uses the clustered index representation.

A second benefit, which is very important, is that it is recalculated for every modification of the underlying data. If you need up to the second aggregations extremely fast, it is better than managing copying data using a trigger. This can also be a detriment, depending on how busy the server is as the aggregations are done synchronously, meaning other users may need to wait for locks to be released.

Finally, and the benefit that can be the biggest help is that, when using Enterprise Edition, SQL Server considers using the aggregates that are stored in your view for queries that look like the query, but doesn't reference the view directly. Getting this to work depends on the needs being fairly limited. The limitations are pretty stiff. For example, a few common bits of coding syntax that are not allowed:

- SELECT * syntax—columns must be explicitly named
- UNION, EXCEPT, or INTERSECT
- Subqueries
- Outer joins or recursive join back to the same table

- TOP in the SELECT clause
- DISTINCT
- SUM() function referencing more than one column
- Almost any aggregate function against an expression that can return NULL
- Reference any other views, or use CTEs or derived tables
- Reference any nondeterministic functions
- Reference data outside the database.
- COUNT(*) – Must use COUNT_BIG(*)
- View not specifying SCHEMABINDING

Need More Review? More on indexed reviews

For more information about indexed views, and a complete list of limitations, read this article in MSDN: *https://msdn.microsoft.com/en-us/library/ms191432.aspx*.

In Listing 1-12, we create a view in the WideWorldImporters database that a customer needed. It is pretty typical, and gives the sum of the cost of what they have purchased, the profit, and the number of line items.

LISTING 1-12 Typical VIEW object a customer may want to view some data

Click here to view code image

```
CREATE VIEW Sales.InvoiceCustomerInvoiceAggregates
WITH SCHEMABINDING
AS
SELECT Invoices.CustomerId,
       SUM(ExtendedPrice * Quantity) AS SumCost,
       SUM(LineProfit) AS SumProfit,
       COUNT_BIG(*) AS TotalItemCount
FROM  Sales.Invoices
          JOIN Sales.InvoiceLines
                ON  Invoices.InvoiceID =
InvoiceLines.InvoiceID
GROUP  BY Invoices.CustomerID;
```

Run the following statement:

Click here to view code image

```
SELECT *
FROM   Sales.InvoiceCustomerInvoiceAggregates;
```

And, checking the plan, you see the plan as shown in Figure 1-22.

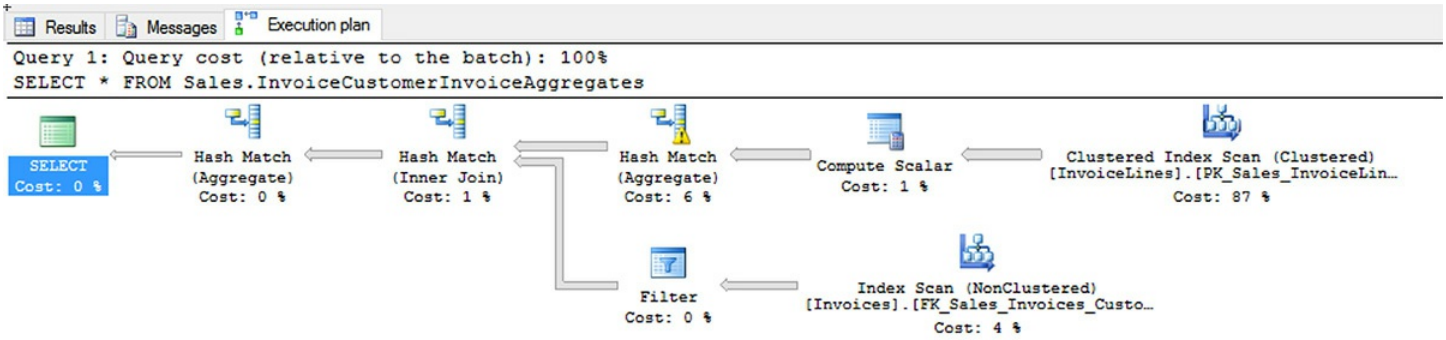


FIGURE 1-22 Query plan from using the Sales.InvoiceCustomerInvoiceAggregates view before indexing

Add the following unique clustered index. It doesn't have to be unique, but if the data allows it, it should be. If duplicated data occurs in the source you receive an error (this is a way to bolt on a uniqueness constraint without changing the DML of a table).

Click here to view code image

```
CREATE UNIQUE CLUSTERED INDEX
XPKInvoiceCustomerInvoiceAggregates on
                    Sales.InvoiceCustomerInvoiceAggregates(Cu
```

Then perform the same query of all data, and the plan changes to what is shown in Figure 1-23. You should also notice the query runs much faster, even with such a small data set.

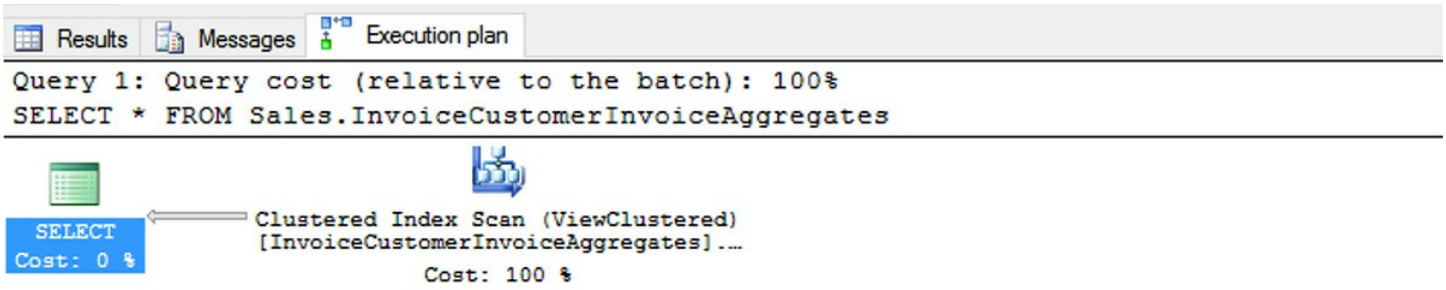


FIGURE 1-23 Query plan from using the Sales.InvoiceCustomerInvoiceAggregates view after adding index

As an example of how this feature can be used to speed up a system without changing the data, look at the plan of the following query:

Click here to view code image

```
SELECT Invoices.CustomerId,
       SUM(ExtendedPrice * Quantity) / SUM(LineProfit),
       COUNT(*) AS TotalItemCount
FROM  Sales.Invoices
          JOIN Sales.InvoiceLines
```

```
                    ON  Invoices.InvoiceID =
InvoiceLines.InvoiceID
GROUP  BY Invoices.CustomerID;
```

It uses the view that we created, because the building blocks used SUM(LineProfit), COUNT(*), and SUM(ExtendedPrice * Quantity) exist in the indexed view. In the plan, you see the same Clustered Index Scan operator, with two Compute Scalars, for the columns that are being output. Indexed views can be a useful tool to apply when you are dealing with a view that is costly and all or some of it can be put into an indexed view. Using indexed views in this manner is a niche use, but it is definitely possible and very powerful when needed. The typical use of indexed views is for reporting, and typically reporting in a reporting database where lots of data modifications are not taking place.

Skill 1.4: Implement columnstore indexes

This final topic is different than what we have covered so far, but it will certainly be on the exam. We have focused primarily on OLTP table design and optimization using the technologies in SQL Server that have been around, in some fashion, since the early versions of the product. Most SQL Server professional developers and DBAs have created tables and applied indexes to them, clustered and non-clustered.

However, our focus now will be centered squarely on reporting, using this new technology in columnstore indexes. Columnstore indexes have changed considerably in each edition of SQL Server since they were introduced. While the basic internal structures are very similar to what was in SQL Server 2012 or 2014, their usage patterns have changed considerably.

Exam Tip

Be sure that if you have used columnstore indexes in earlier editions of the product that you review this topic in detail.

This section covers how to:

- Determine use cases that support the use of columnstore indexes
- Identify proper usage of clustered and non-clustered columnstore indexes
- Design standard non-clustered indexes in conjunction with clustered columnstore indexes
- Implement columnstore index maintenance

Determine use cases that support the use of columnstore indexes

Columnstore indexes are purpose built for reporting scenarios, particularly when dealing

with large quantities of data. Columnstore indexes are based on the concept of a *columnar database*, of which the concept is not a new one (if you would like a deeper explanation of columnar databases, the following paper provides as much and more than you may want: *http://db.csail.mit.edu/pubs/abadi-column-stores.pdf*). The base idea is that instead of storing all of the data for a row together, you store all of the data for a column together, as shown conceptually in Figure 1-24. Each column is stored independently, but the rows of the table are kept in the same order in each segment.

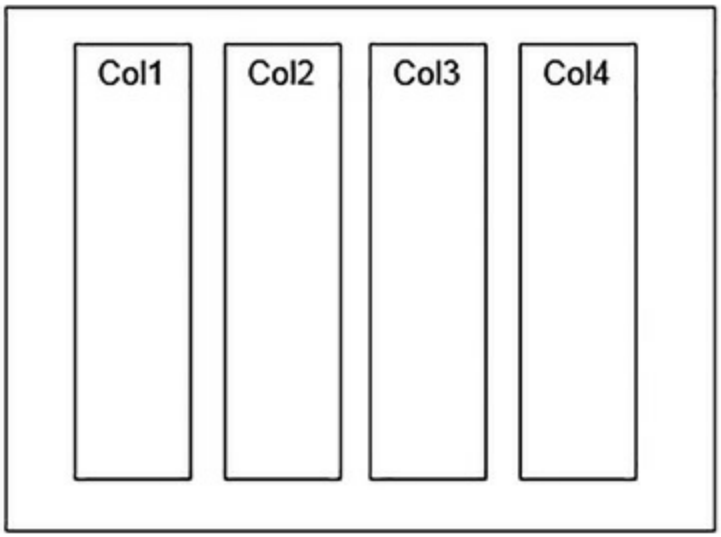


FIGURE 1-24 Conceptual format of a columnar database

This format is particularly apt when you only need a small percentage of the columns from the table, particularly when you need a large percentage of the rows of the table. For example, a query of the format SELECT SUM(Col1) FROM TableName; would only need to scan the structure for Col1, and would never need to touch Col2, Col3, or Col4.

Column-oriented indexes, because they are not ordered, are not useful for picking only a few rows out of a table, so the implementation of columnstore indexes before SQL Server 2016 was not tremendously flexible in how it might be applied. Another limitation is that there are several data types that are not supported:

- varchar(max) and nvarchar(max)
- rowversion (also known as timestamp)
- sql_variant
- CLR based types (hierarchyid and spatial types)
- xml
- ntext, text, and image (though rightfully so as these data types have been deprecated for some time)

In SQL Server 2016, you can apply these indexes not only to tables that are only for strictly for reporting, but also to tables in your live OLTP database for real-time analytics.

While the maintenance of these indexes is more costly and complex than rowstore indexes, one columnstore index can replace almost all of the indexes you may have applied for analytics.

Figure 1-25 is a conceptual drawing of the structure of a columnstore index as implemented by SQL Server. Each *row group* contains up to 1,048,576 rows each, broken down into *segments* that are all ordered physically the same, though in no logical order.

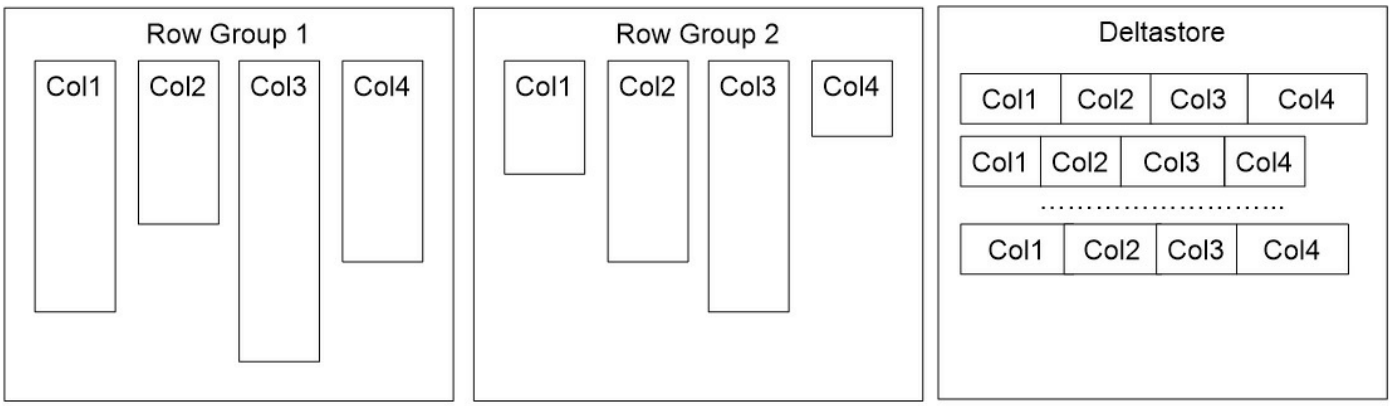


FIGURE 1-25 Conceptual format of a columnstore index

In each row group, there is a set of *column segments*, that store the data for one single column. In Figure 1-25, note that the column segments are drawn as differently sized, because each of the segments is compressed, using similar constructs like can be done with page compression on classic row oriented structures, but instead of an 8K page, compression can take place over the single row group, or even all row groups for far greater compression. At the structure and segment, data is compressed using a process like normalization, where values that are duplicated over and over are replaced by a smaller value to look up the actual value.

Each of the segments have information stored about the values to let the query processor know if the segment can be skipped in processing, so if you have a WHERE clause that is looking for Col1 > 100, and the max value is 50, the segment can be skipped.

The *deltastore* structure comes into play when you are modifying the data in a table with a columnstore index. New rows are placed into the deltastore in a heap structure until the rows in the deltastore are compressed, and moved into a compressed row group in column segments. DELETE operations simply mark the row as removed from the column segment, telling the query processor to ignore the row. UPDATE operations in the columnstore index are a delete from the columnstore index, and then the row is added to the deltastore like a normal INSERT operation.

The process that moves rows from the deltastore to compressed rowgroups is called the *tuple mover*. It is a background process that runs periodically and does most of the management of your columnstore index maintenance. However, depending on how you use the table with the columnstore index, you need to do some maintenance as well (this is covered in the last section of the chapter: "Implement Columnstore Index Maintenance").

Need More Review? More Information about columnstore indexes

The introduction to columnstore indexes that is provided in this section has been strictly as a review of key concepts about how the indexes work. There is much more useful information in the MSDN: Columnstore Indexes Guide *https://msdn.microsoft.com/en-us/library/gg492088.aspx*. It contains links and references to a lot of really great information, far more than we are able to provide in this chapter.

Note More on CREATECOLUMNSTOREINDX

Another good resource to read over is the CREATE COLUMNSTORE INDEX documentation here: *https://msdn.microsoft.com/en-us/library/gg492153.aspx* as it contains any other limitations that you should familiar yourself with.

Identify proper usage of clustered and non-clustered columnstore indexes

Much like rowstore indexes, the distinction between clustered and non-clustered indexes is whether it is a separate structure, or if it changes the physical structure of the table. A clustered columnstore index compresses the base rows, removing the row based storage altogether. The non-clustered columnstore leaves the heap or clustered rowstore index and adds a separate compressed structure. At a high level, columnstore indexes support two scenarios, one of which works with each type of index:

- **Dimensional formatted data warehouses (Clustered Columnstore Indexes)** Different than relational databases we have covered so far, we look at the pattern of how dimensional databases are structured, and how these indexes work well with them
- **Analytics on OLTP tables (Nonclustered Columnstore Indexes)** Due to how these indexes are maintained, it can be that when operational reporting is needed, a columnstore index performs better overall than a B-Tree index, particularly if multiple complex B-Tree indexes are needed, as the query processor only needs to synchronously maintain one analytical index.

These indexes are not used to improve performance of small row by row operations, but rather when one needs to work through large sets of data, touching most of the rows. In this section, we review how this need applies to the two identified scenarios, and how they differ calling for the different type of index.

One important thing about any use case for columnstore indexes is that they should have a large amount of data. As we have noted, the optimum amount of data in a row group is 1,048,576. If your tables only have a few thousand rows (or even a few hundred thousand

rows) in them, columnstore indexes may not be what you need, though they can still be applied and used.

Using clustered columnstore indexes on dimensional data warehouse structures

A data warehouse can mean many things to people, but one of the primary meanings is based on the pattern of a *star schema*. The following is a brief review of a star schema from the WideWordImportersDW sample database that is a companion to the WideWorldImporters sample database that we have been using so far for performance examples. The name star schema comes from the way a data model looks when the structure is implemented as shown in Figure 1-26.

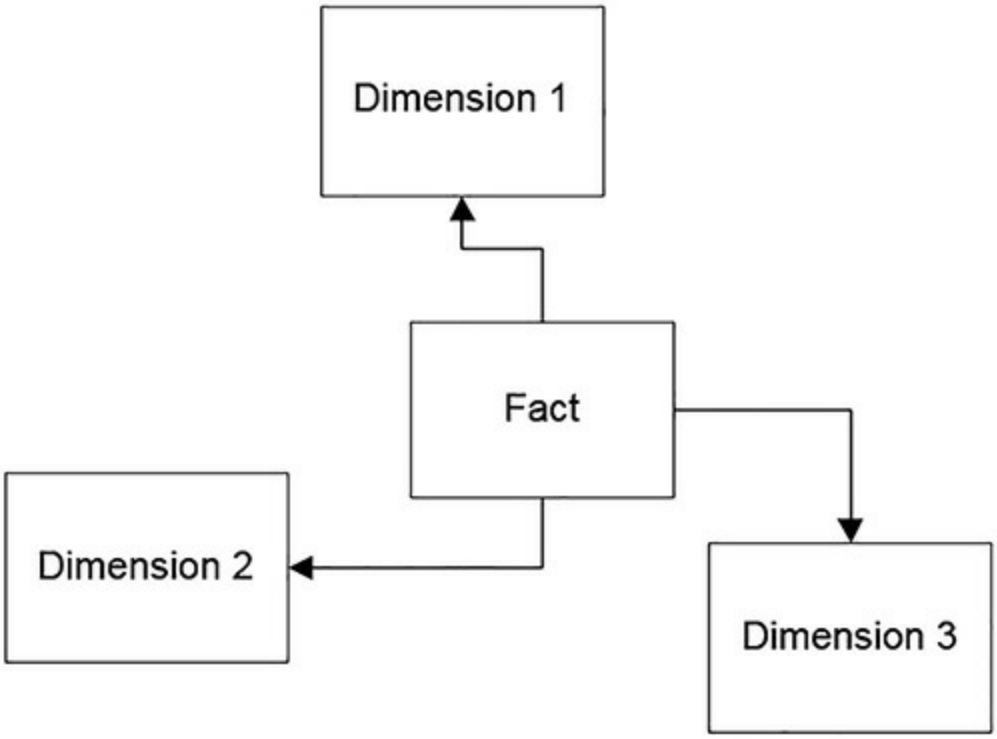


FIGURE 1-26 Conceptual format of star schema

In some cases, a dimension links to other dimensions forming what is referred to as a *snowflake schema*, though ideally there is one join between fact and dimension. The concept of a star schema is that there is one central table that contains measurements (called a *fact table*) that needs to be reported on (typically the goal is to perform some aggregate), and a set of foreign key values that link to tables of values that the data can be summarized by (called *dimensions*). One such example in the WideWorldImportersDW is the Fact.[Order] table, shown in Listing 1-13.

LISTING 1-13 Columns in the Fact.Order table in WideWorldImportersDW

Click here to view code image

```
CREATE TABLE Fact.[Order]
(
```

```
    [Order Key] bigint IDENTITY(1,1) NOT NULL,
    [City Key] int NOT NULL,
    [Customer Key] int NOT NULL,
    [Stock Item Key] int NOT NULL,
    [Order Date Key] date NOT NULL,
    [Picked Date Key] date NULL,
    [Salesperson Key] int NOT NULL,
    [Picker Key] int NULL,
    [WWI Order ID] int NOT NULL,
    [WWI Backorder ID] int NULL,
    [Description] nvarchar(100) NOT NULL,
    [Package] nvarchar(50) NOT NULL,
    [Quantity] int NOT NULL,
    [Unit Price] decimal(18, 2) NOT NULL,
    [Tax Rate] decimal(18, 3) NOT NULL,
    [Total Excluding Tax] decimal(18, 2) NOT NULL,
    [Tax Amount] decimal(18, 2) NOT NULL,
    [Total Including Tax] decimal(18, 2) NOT NULL,
    [Lineage Key] int NOT NULL
);
```

Breaking this table down, the [Order Key] column is a surrogate key. Column: [City Key] down to [Picker Key] are *dimension keys*, or dimension foreign key references. The cardinality of the dimension compared to the fact table is generally very low. You could have millions of fact rows, but as few as 2 dimension rows. There are techniques used to combine dimensions, but the most germane point to our discussion of columnstore indexes is that dimensions are lower cardinality tables with factors that one might group the data. Sometimes in data warehouses, FOREIGN KEY constraints are implemented, and sometimes not. Having them in the database when querying can be helpful, because they provide guidance to tools and the optimizer. Having them on during loading can hinder load performance.

Columns from [WWI BackorderID] to [Package] are referred to as *degenerate dimensions*, which means they are at, or are nearly at, the cardinality of the row and are more often used for finding a row in the table, rather than for grouping data.

Columns from [Quantity] down to [Total Including Tax] as called *measures*. These are the values that a person writing a query applies math to. Many measures are additive, meaning you can sum the values (such as [Quantity] in this example, and others are not, such as [Tax Rate]. If you add a 10 percent tax rate to a 10 percent tax rate, you don't get 20 percent, no matter your political affiliations.

The [Lineage Key] is used to track details of where data comes from during loads. The table Integration.Lineage contains information about what was loaded and when. In Listing 1-14, is the basic code for two dimensions that relate to the Fact.Orders table.

LISTING 1-14 Columns in the Customer and Date dimensions in WideWorldImportersDW

Click here to view code image

```
CREATE TABLE Dimension.Customer
(
    [Customer Key] int NOT NULL,
    [WWI Customer ID] int NOT NULL,
    [Customer] nvarchar(100) NOT NULL,
    [Bill To Customer] nvarchar(100) NOT NULL,
    [Category] nvarchar(50) NOT NULL,
    [Buying Group] nvarchar(50) NOT NULL,
    [Primary Contact] nvarchar(50) NOT NULL,
    [Postal Code] nvarchar(10) NOT NULL,
    [Valid From] datetime2(7) NOT NULL,
    [Valid To] datetime2(7) NOT NULL,
    [Lineage Key] int NOT NULL
);
CREATE TABLE Dimension.Date(
   Date date NOT NULL,
    [Day Number] int NOT NULL,
    [Day] nvarchar(10) NOT NULL,
    [Month] nvarchar(10) NOT NULL,
    [Short Month] nvarchar(3) NOT NULL,
    [Calendar Month Number] int NOT NULL,
    [Calendar Month Label] nvarchar(20) NOT NULL,
    [Calendar Year] int NOT NULL,
    [Calendar Year Label] nvarchar(10) NOT NULL,
    [Fiscal Month Number] int NOT NULL,
    [Fiscal Month Label] nvarchar(20) NOT NULL,
    [Fiscal Year] int NOT NULL,
    [Fiscal Year Label] nvarchar(10) NOT NULL,
    [ISO Week Number] int NOT NULL
);
```

We won't go into too much detail about all of these columns in the tables. But, the [Customer Key] and the Date columns are the columns that are referenced from the fact table. In the Dimensions.Customer table, the [Valid From] and [Valid To] columns set up a *slowly changing dimension*, where you could have multiple copies of the same customer over time, as attributes change. There are no examples of having multiple versions of a customer in the sample database, and it would not change our indexing example either.

Note More on fact tables

Fact tables are generally designed to be of a minimal width, using integer types for foreign key values, and very few degenerate dimensions if at all

possible. For demos, the cost savings you see could be fairly small. However, in a real fact table, the number of rows can be very large, in the billions or more, and the calculations attempted more complex than just straightforward aggregations.

All of the other columns in the dimensions (other than [Lineage Key], which provides the same sort of information as for the fact) can be used to group data in queries. Because the WideWorldImporterDW database starts out configured for examples, we can begin by dropping the columnstore index that is initially on all of the fact tables.

Click here to view code image

```
DROP INDEX [CCX_Fact_Order] ON [Fact].[Order];
```

The table starts out with indexes on all of the foreign keys, as well as primary keys on the dimension keys that the query uses. Perform the following query (there are 231,412 rows in the Fact.[Order] table), which you likely note runs pretty quickly without the columnstore index):

Click here to view code image

```
SELECT Customer.Category, Date.[Calendar Month Number],
        COUNT(*) AS SalesCount,
        SUM([Total Excluding Tax]) as SalesTotal
FROM   Fact.[Order]
         JOIN Dimension.Date
            ON Date.Date = [Order].[Order Date Key]
         JOIN Dimension.Customer
            ON Customer.[Customer Key] = [Order].[Customer Key]
GROUP BY Customer.Category, Date.[Calendar Month Number]
ORDER BY Category, Date.[Calendar Month Number], SalesCount,
SalesTotal;
```

The plan for this query, shown in Figure 1-27 is complicated by the cost of scanning the table, which pushes the query to use parallelism, even on my VM. The largest cost is the table scan of the heap structure that was left after removing the clustered columnstore index.

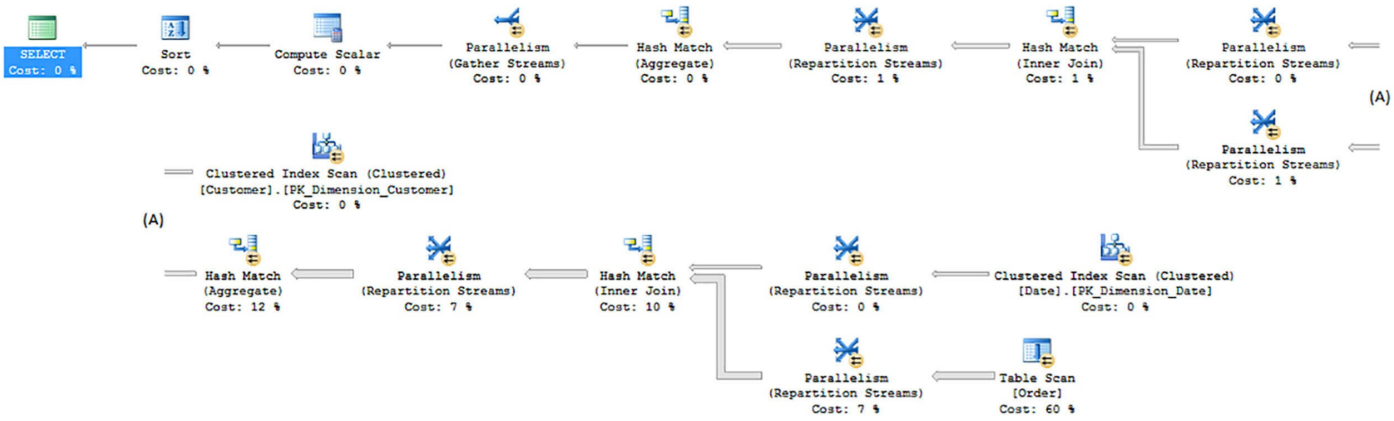


FIGURE 1-27 Plan of the basic data warehousing style query, without a columnstore index

Figure 1-27 has the following output:

Click here to view code image

```
Table 'Customer'. Scan count 3, logical reads 40, physical
reads 0, read-ahead reads 0,
lob logical reads 0, lob physical reads 0,

lob read-ahead reads 0.

Table 'Date'. Scan count 3, logical reads 79, physical reads 0,
read-ahead reads 0, lob
logical reads 0, lob physical reads 0,

lob read-ahead reads 0.

Table 'Order'. Scan count 7, logical reads 5908, physical reads
0, read-ahead reads 0,
lob logical reads 0, lob physical reads 0,

lob read-ahead reads 0.

Table 'Workfile'. Scan count 0, logical reads 0, physical reads
0, read-ahead reads 0,
lob logical reads 0, lob physical reads 0,

lob read-ahead reads 0.

Table 'Worktable'. Scan count 0, logical reads 0, physical
reads 0, read-ahead reads 0,
lob logical reads 0, lob physical reads 0,

lob read-ahead reads 0.
```

```
CPU time = 344 ms,  elapsed time = 276 ms.
```

Most of the plan is typical, as you often see Hash Match operators when joining two larger sets of data, which could not realistically be ordered in the same order as one another. Even with the smallish table structure for the fact table, there are 5908 logical reads (which is the same number of reads if it scanned the entire table once).

Prior to columnstore indexes, a suggested index to help this query would have been to use a covering index to cover the needs of this query so you didn't have to touch any data other than the query needed. The optimizer suggested such an index for our query:

Click here to view code image

```
CREATE NONCLUSTERED INDEX SpecificQuery ON [Fact].[Order]
([Customer Key])
INCLUDE ([Order Date Key],[Total Excluding Tax]);
```

After adding this suggested index, the plan for this query is very similar, without the parallelism, and instead of a Table Scan operator that is 60 percent of the cost, there is an index scan that is 23 percent. The logical reads are reduced to 871 instead of 5908. The processing still takes around 300 ms, and actually took a bit longer than the full table scan versions at times. The problem with indexes that are tailored to specific queries is, if you want to add another column to your query, this index stops being of value. Columnstore indexes basically give you great aggregate and scan performance for most of the combinations of attributes you might consider without custom pre-planning.

Now, add the clustered columnstore index back to the table.

Click here to view code image

```
CREATE CLUSTERED COLUMNSTORE INDEX [CCX_Fact_Order] ON [Fact].
[Order];
```

As the name clustered implies, this changes the internal structure of the table to be the columnar structures. We did not remove any of the rowstore indexes, and we review why you would or would not want to use both in tandem in section later in this chapter entitled "Design standard non-clustered indexes in conjunction with clustered columnstore indexes".

The row locator for the rowstore indexes has been changed from the physical location in the heap, to the position in the columnstore structure (the row group, and the position in the row group). It is a bit more complex than this, and if you want more information, Niko Neugebauer has a great article about it here: *http://www.nikoport.com/2015/09/06/columnstore-indexes-part-65-clustered-columnstore-improvements-in-sql-server-2016/*.

For nearly all data warehousing applications, the clustered columnstore is a useful structure for fact tables when the table is large enough. Since the main copy of the data is compressed, you can see very large space savings, even having the table be 10 percent of

the original size. Couple this with the usual stability of data in a data warehouse, with minimal changes to historical data, make the clustered columnstore typically ideal. Only cases where something does not work, like one of the data types that were mentioned in the introductory section (varchar(max) or nvarchar(max), for example) would you likely want to consider using a non-clustered columnstore index.

Whether or not a clustered columnstore index will be useful with a dimension will come down to how it is used. If the joins in your queries do not use a Nested Loop operator, there is a good chance it could be useful.

Perform the query again, and check the plan shown in Figure 1-28, which shows a tremendous difference:

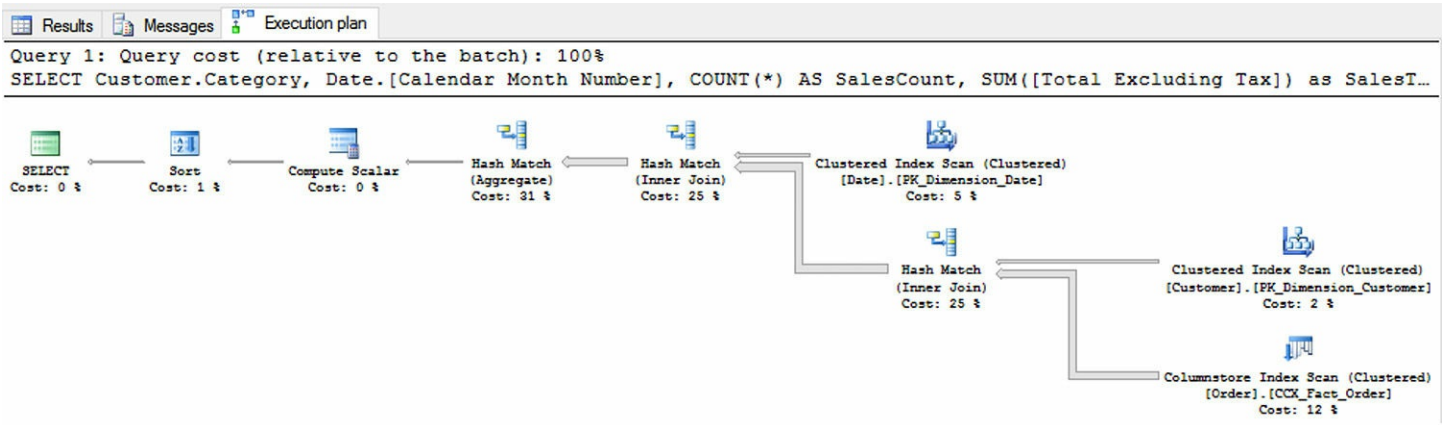


FIGURE 1-28 Plan of the basic data warehousing style query, after adding a columnstore index

Figure 1-28 has the following output:

Click here to view code image

```
Table 'Order'. Scan count 1, logical reads 0, physical reads 0,
read-ahead reads 0, lob
logical reads 256, lob physical reads 0,

lob read-ahead reads 0.

Table 'Order'. Segment reads 4, segment skipped 0.

Table 'Worktable'. Scan count 0, logical reads 0, physical
reads 0, read-ahead reads 0,
lob logical reads 0, lob physical reads 0,

lob read-ahead reads 0.

Table 'Customer'. Scan count 1, logical reads 15, physical
reads 0, read-ahead reads 0,
lob logical reads 0, lob physical reads 0,
```

```
lob read-ahead reads 0.

Table 'Date'. Scan count 1, logical reads 28, physical reads 0,
read-ahead reads 0, lob
logical reads 0, lob physical reads 0,

lob read-ahead reads 0.

CPU time = 15 ms,  elapsed time = 65 ms.
```

The logical reads are down to 256 in the lob reads for the segments, since the column segments are stored in a form of large varbinary storage. Note, too, that it took just 68ms rather than 286.

One thing that makes columnstore indexes better for queries such as those found in data warehouses is *batch execution mode*. When the query processor is scanning data in the columnstore index, it is possible for it to process rows in chunks of 900 rows at a time, rather than one row at a time in the typical row execution mode. Figure 1-29 displays the tool tip from hovering over the Columnstore Index Scan operator from Figure 1-28. The third and fourth lines down show you the estimated and actual execution mode. Batch execution mode can provide great performance improvements.

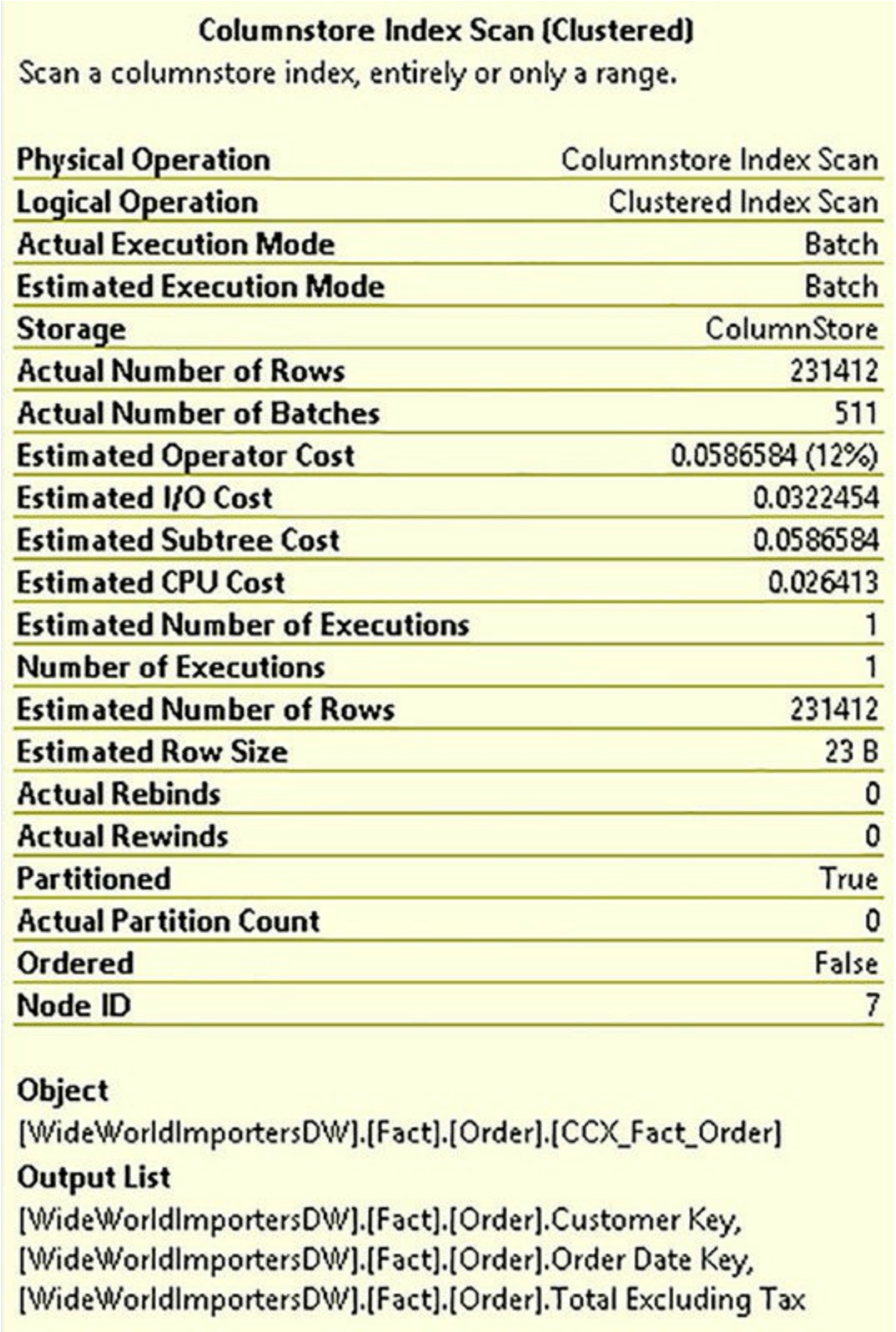


FIGURE 1-29 Tooltip showing Columnstore Index Scan operator using Batch Execution Mode

Finally, just for comparison, let us drop the clustered columnstore index, and add a non-clustered columnstore index. When you are unable to use a clustered one due to some limitation, non-clustered columnstore indexes are just as useful to your queries, but the base table data is not compressed, giving you less overall value.

In our demo, include all of the columns except for [Lineage ID] and [Description], which have no real analytic value to our user:

Click here to view code image

```
CREATE NONCLUSTERED COLUMNSTORE INDEX [NCCX_Fact_Order] ON
[Fact].[Order] (
       [Order Key] ,[City Key] ,[Customer Key] ,[Stock Item
Key]
      ,[Order Date Key] ,[Picked Date Key] ,[Salesperson Key] ,
[Picker Key]
      ,[WWI Order ID],[WWI Backorder ID],[Package]
      ,[Quantity],[Unit Price],[Tax Rate],[Total Excluding Tax]
      ,[Tax Amount],[Total Including Tax]);
```

Executing the query one more time, the plan looks exactly like the query did previously, other than it is using a non-clustered columnstore operator rather than a clustered one. The number of reads go up slightly in comparison to the clustered example, but not tremendously. The beauty of the columnstore indexes however is how well they adapt to the queries you are executing. Check the plan and IO/time statistics for the following query, that adds in a new grouping criteria, and a few additional aggregates:

Click here to view code image

```
SELECT Customer.Category, Date.[Calendar Year],
        Date.[Calendar Month Number],
        COUNT(*) as SalesCount,
        SUM([Total Excluding Tax]) AS SalesTotal,
        AVG([Total Including Tax]) AS AvgWithTaxTotal,
        MAX(Date.Date) AS MaxOrderDate
FROM   Fact.[Order]
         JOIN Dimension.Date
            ON Date.Date = [Order].[Order Date Key]
         JOIN Dimension.Customer
            ON Customer.[Customer Key] = [Order].[Customer Key]
GROUP BY Customer.Category, Date.[Calendar Year], Date.
[Calendar Month Number]
ORDER BY Category, Date.[Calendar Month Number], SalesCount,
SalesTotal;
```

You should see very little change, including the time required to perform the query. This ability to cover many analytical indexing needs is what truly makes the columnstore indexes a major difference when building data warehouse applications. Hence, both the clustered and non-clustered columnstore indexes can be used to greatly improve your data warehouse loads, and in a later section, we review some of the differences.

Need More Review? Using columnstore indexes in data warehousing

For more information about using columnstore indexes for data warehousing scenarios, the following page in MSDN's Columnstore Indexes Guide has more information: *https://msdn.microsoft.com/en-us/library/dn913734.aspx*.

Using non-clustered columnstore indexes on OLTP tables for advanced analytics

The typical data warehouse is refreshed daily, as the goal of most analytics is to take some amount of past performance and try to replicate and prepare for it. "We sold 1000 lunches on average on Tuesdays following a big game downtown, and we have 500 plates, so as a company, we need to plan to have more in stock." However, there are definitely reports that need very up to date data. "How many lunches have we sold in the past 10 minutes? There are 100 people in line." At which point, queries are crafted to use the OLTP database.

By applying a non-clustered columnstore index to the table you wish to do real-time analytics on, you can enable tremendous performances with little additional query tuning. And depending on your concurrency needs, you can apply a few settings to tune how the columnstore index is maintained.

Note Memory optimized tables

Memory optimized tables, which is covered in Skill 3.4, can also use columnstore indexes. While they are called clustered, and they must have all of the columns of the table; they are more similar in purpose and usage to non-clustered columnstore indexes because they do not change the physical storage of the table.

Columnstore indexes can be used to help greatly enhance reporting that accesses an OLTP database directly, certainly when paired with concurrency techniques that we cover in Chapter 3. Generally speaking, a few questions need to be considered: "How many reporting queries do you need to support?" and "How flexible does the reporting need to be?"

If, for example, the report is one, fairly rigid report that uses an index with included columns to cover the needs of that specific query could be better. But if the same table supports multiple reports, and particularly if there needs to be multiple indexes to support analytics, a columnstore index is a better tool.

In the WideWorldImporters database, there are a few examples of tables that have a non-clustered columnstore index, such as the OrderLines table, the abbreviated DDL of which is shown in Listing 1-15.

LISTING 1-15 Abbreviated structure of the WideWorldImporters.Sales.InvoiceLines table with non-clustered columnstore index

Click here to view code image

```
CREATE TABLE Sales.InvoiceLines
(
    InvoiceLineID int NOT NULL,
    InvoiceID int NOT NULL,
    StockItemID int NOT NULL,
    Description nvarchar(100) NOT NULL,
    PackageTypeID int NOT NULL,
    Quantity int NOT NULL,
    UnitPrice decimal(18, 2) NULL,
    TaxRate decimal(18, 3) NOT NULL,
    TaxAmount decimal(18, 2) NOT NULL,
    LineProfit decimal(18, 2) NOT NULL,
    ExtendedPrice decimal(18, 2) NOT NULL,
    LastEditedBy int NOT NULL,
    LastEditedWhen datetime2(7) NOT NULL,
 CONSTRAINT PK_Sales_InvoiceLines PRIMARY KEY
              CLUSTERED ( InvoiceLineID )
 );
--Not shown: FOREIGN KEY constraints, indexes other than the PK

CREATE NONCLUSTERED COLUMNSTORE INDEX NCCX_Sales_OrderLines ON
Sales.OrderLines
(
    OrderID,
    StockItemID,
    Description,
    Quantity,
    UnitPrice,
    PickedQuantity
) ON USERDATA;
```

Now, if you are reporting on the columns that are included in the columnstore index, only the columnstore index is used. The needs of the OLTP (generally finding and operating on just a few rows), are served from the typical rowstore indexes. There are a few additional ways to improve the utilization and impact of the columnstore index on the overall performance of the table, which we examine in the following sections:

- Targeting analytically valuable columns only in columnstore
- Delaying adding rows to compressed rowgroups
- Using filtered non-clustered columnstore indexes to target hot data

Need More Review? Using columnstore indexes for real-time analytics

In addition to the tips covered in the text, there is more detail in the following

MSDN Article called “Get Started with Columnstore for real time operational analytics:” *https://msdn.microsoft.com/en-us/library/dn817827.aspx*.

Targeting analytically valuable columns only in columnstore

As shown with the columnstore index that was created in the Sales.OrderLines table, only certain columns were part of the non-clustered columnstore index. This can reduce the amount of data duplicated in the index (much like you would usually not want to create a rowstore index with every column in the table as included columns), reducing the required amount of maintenance.

Delaying adding rows to compressed rowgroups

Columnstore indexes have to be maintained in the same transaction with the modification statement, just like normal indexes. However, modifications are done in a multi-step process that is optimized for the loading of the data. As described earlier, all modifications are done as an insert into the delta store, a delete from a column segment or the delta store, or both for an update to a row. The data is organized into compressed segments over time, which is a burden in a very busy system. Note that many rows in an OLTP system can be updated multiple times soon after rows are created, but in many systems are relatively static as time passes.

Hence there is a setting that lets you control the amount of time the data stays in the deltastore. The setting is: COMPRESSION_DELAY, and the units are minutes. This says that the data stays in the delta rowgroup for at least a certain number of minutes. The setting is added to the CREATE COLUMNSTORE INDEX statement, as seen in Listing 1-16.

LISTING 1-16 Changing the non-clustered columnstore index to have COMPRESSION_DELAY = 5 minutes

Click here to view code image

```
CREATE NONCLUSTERED COLUMNSTORE INDEX NCCX_Sales_OrderLines ON
Sales.OrderLines
(
          OrderID,
          StockItemID,
          Description,
          Quantity,
          UnitPrice,
          PickedQuantity
) WITH (DROP_EXISTING = ON, COMPRESSION_DELAY = 5) ON USERDATA;
```

Now, in this case, say the PickedQuantity is important to the analytics you are trying to

perform, but it is updated several times in the first 5 minutes (on average) after the row has been created. This ensures that the modifications happens in the deltastore, and as such does not end up wasting space in a compressed rowgroup being deleted and added over and over.

Using filtered non-clustered columnstore indexes to target colder data

Similar to filtered rowstore indexes, non-clustered columnstore indexes have filter clauses that allow you to target only data that is of a certain status. For example, Listing 1-17 is the structure of the Sales.Orders table. Say that there is a business rule that once the items have been picked by a person, it is going to be shipped. Up until then, the order could change in several ways. The user needs to be able to write some reports on the orders that have been picked.

LISTING 1-17 Base structure of the Sales.Orders Table

Click here to view code image

```
CREATE TABLE Sales.Orders
(
    OrderID int NOT NULL,
    CustomerID int NOT NULL,
    SalespersonPersonID int NOT NULL,
    PickedByPersonID int NULL,
    ContactPersonID int NOT NULL,
    BackorderOrderID int NULL,
    OrderDate date NOT NULL,
    ExpectedDeliveryDate date NOT NULL,
    CustomerPurchaseOrderNumber nvarchar(20) NULL,
    IsUndersupplyBackordered bit NOT NULL,
    Comments nvarchar(max) NULL,
    DeliveryInstructions nvarchar(max) NULL,
    InternalComments nvarchar(max) NULL,
    PickingCompletedWhen datetime2(7) NULL,
    LastEditedBy int NOT NULL,
    LastEditedWhen datetime2(7) NOT NULL,
    CONSTRAINT PK_Sales_Orders PRIMARY KEY CLUSTERED
    (
       OrderID ASC
    )
);
```

One could then, applying a few of the principles we have mentioned in these sections, choose only the columns we are interested in, though we should not need to add a compression delay for this particular case since once the PickedByPersonID is set, we are

saying the data is complete.

So we might set up:

Click here to view code image

```
CREATE NONCLUSTERED COLUMNSTORE INDEX NCCI_Orders ON
Sales.Orders
(
      PickedByPersonId,
      SalespersonPersonID,
      OrderDate,
      PickingCompletedWhen
)
WHERE PickedByPersonId IS NOT NULL;
```

One additional thing you can do, if you need your reporting to span the cold and hot data, and that is to cluster your data on the key that is use for the filtering. So in this case, if you clustered your table by PickedByPersonId, the optimizer would easily be able to split the set for your queries. This could seem counter to the advice given earlier about clustering keys and it generally is. However, in some cases this could make a big difference if the reporting is critical. It is covered in more detail by Sunil Agarwal in his blog here (*https://blogs.msdn.microsoft.com/sqlserverstorageengine/2016/03/06/real-time-operational-analytics-filtered-nonclustered-columnstore-index-ncci/*) when he suggested using the a column with a domain of order status values in his example to cluster on, even though it has only 6 values and the table itself has millions.

Design standard non-clustered indexes in conjunction with clustered columnstore indexes

When using columnstore indexes in your database solution, it is important to know what their values and detriments are. To review, here are some of the attributes we have discussed so far:

- **Columnstore Indexes** Great for working with large data sets, particularly for aggregation. Not great for looking up a single row, as the index is not ordered
  - **Clustered** Compresses the table to greatly reduce memory and disk footprint of data.
  - **Nonclustered** Addition to typical table structure, ideal when the columns included cover the needs of the query.
- **Rowstore Indexes** Best used for seeking a row, or a set of rows in order.
  - **Clustered** Physically reorders table's data in an order that is helpful. Useful for the primary access path where you fetch rows along with the rest of the row data, or for scanning data in a given order.
  - **Nonclustered** Structure best used for finding a single row. Not great for scanning

unless all of the data needed is in the index keys, or is in included in the leaf pages of the index.

If you have worked with columnstore indexes in SQL Server 2012 or 2014, it is necessary to change how you think about using these indexes. In 2012, SQL Server only had read only non-clustered columnstore indexes, and to modify the data in the index (and any of the rows in the table), the index needed to be dropped and completely rebuilt. In 2014, read/write clustered columnstore indexes were added, but there was no way to have a rowstore index on the same table with them. When you needed to fetch a single row, the query processor needed to scan the entire table. If your ETL did many updates or deletes, the operation was costly. So for many applications, sticking with the drop and recreating a non-clustered index made sense.

In SQL Server 2016, the version that you are studying for in this exam, both types of columnstore indexes are read/write, and both allow you to have complimentary rowstore indexes. In this section, we focus on adding non-clustered indexes to your clustered columnstore indexes, which in the previous section we have established as generally the best practice for data warehousing situations, mostly fact tables, and possibly very large dimensions. The columnstore indexes are there to aid in analytical queries, but there are a few other needs to consider.

To demonstrate, begin by making a copy of the Fact.Sale table in the WideWorldImportersDW database, and adding a clustered columnstore index.

Click here to view code image

```
SELECT *
INTO   Fact.SaleBase
FROM   Fact.Sale;

CREATE CLUSTERED COLUMNSTORE INDEX CColumnsStore ON
Fact.SaleBase;
```

You see that if you perform an aggregation query, the columnstore index is used, and the performance is stellar:

Click here to view code image

```
SELECT Date.[Fiscal Year], Customer.Category,  Sum(Quantity) as
NumSales
FROM   Fact.SaleBase
         JOIN Dimension.Customer
            on Customer.[Customer Key] = SaleBase.[Customer
Key]
         JOIN Dimension.Date
            ON Date.Date = SaleBase.[Invoice Date Key]
GROUP BY Date.[Fiscal Year], Customer.Category
ORDER BY Date.[Fiscal Year], Customer.Category;
```

And the plan is shown in Figure 1-30.

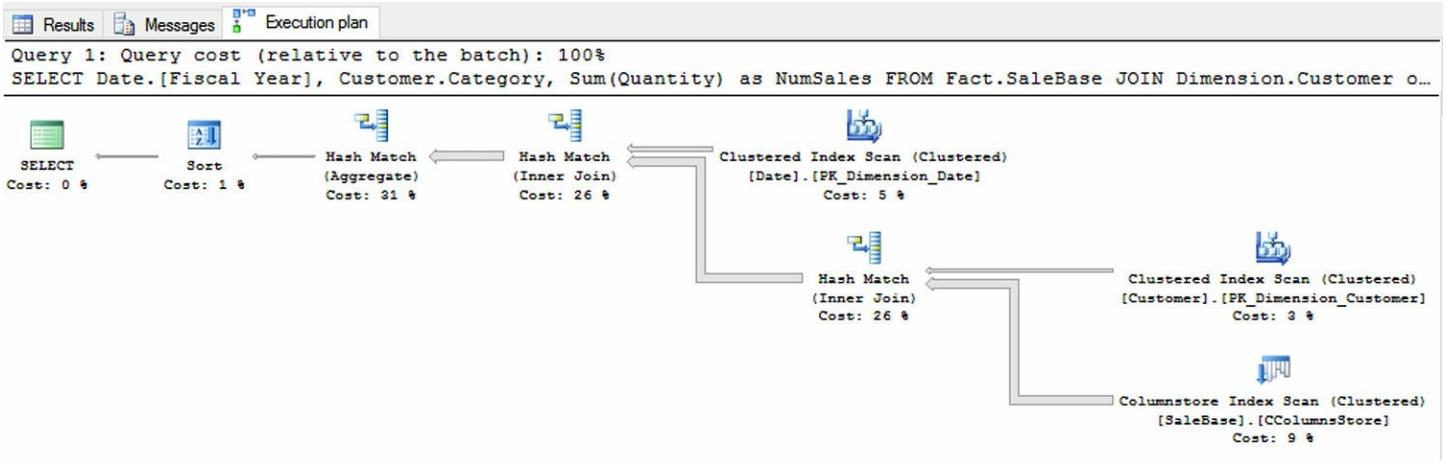


FIGURE 1-30 The plan with the query aggregating data from the fact

Figure 1-30 has the following output:

Click here to view code image

```
Table 'Worktable'. Scan count 0, logical reads 0, physical
reads 0, read-ahead reads 0,
lob logical reads 0, lob physical reads 0,

lob read-ahead reads 0.

Table 'Customer'. Scan count 1, logical reads 15, physical
reads 0, read-ahead reads 0,
lob logical reads 0, lob physical reads 0,

lob read-ahead reads 0.

Table 'Date'. Scan count 1, logical reads 28, physical reads 0,
read-ahead reads 0, lob
logical reads 0, lob physical reads 0,

lob read-ahead reads 0.

CPU time = 15 ms,  elapsed time = 22 ms.
```

What is likely unexpected is what happens when filtering on a single value in a column not yet referenced by adding to the statement the following WHERE clause (a common operation when doing ETL where data can change):

Click here to view code image

```
WHERE SaleBase.[Sale Key] = 26974
```

The plan changes to something that looks better, but actually takes more time and considerably more IO as seen in Figure 1-31.

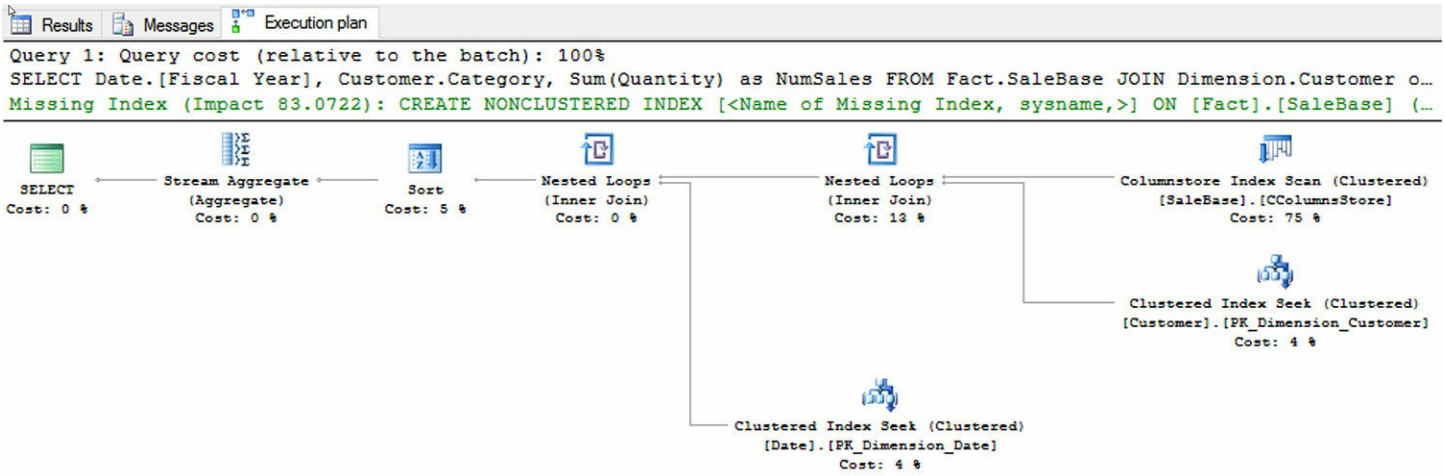


FIGURE 1-31 The plan with the query aggregating one row based on the [Sale Key]

Figure 1-31 has the following output:

Click here to view code image

```
Table 'SaleBase'. Scan count 1, logical reads 0, physical reads
0, read-ahead reads 0,
lob logical reads 347, lob physical reads 0,

lob read-ahead reads 0.

Table 'SaleBase'. Segment reads 1, segment skipped 0.

Table 'Worktable'. Scan count 0, logical reads 0, physical
reads 0, read-ahead reads 0,
lob logical reads 0, lob physical reads 0,

lob read-ahead reads 0.

Table 'Date'. Scan count 0, logical reads 2, physical reads 0,
read-ahead reads 0, lob
logical reads 0, lob physical reads 0,

lob read-ahead reads 0.

Table 'Customer'. Scan count 0, logical reads 2, physical reads
0, read-ahead reads 0,
lob logical reads 0, lob physical reads 0,

lob read-ahead reads 0.

CPU time = 0 ms,  elapsed time = 40 ms.
```

Note that now the major cost is spent in the Columnstore Scan operator, which isn't surprising since the query processor has to touch all of the rows in the table; for the column

segment for the [Sale Key] in any case. This cost isn't too much in this very small fact table (less than 300,000 rows), but it is very telling that you have an issue if you're needing to fetch rows one at a time for some reason, either to update data, or delete a row.

So next we add indexes to the table for any cases where you want to access rows one at a time. Examples in the Fact.SaleBase table are columns like the surrogate key: [Sale Key], and the degenerate key: [WWI Invoice ID]. Other uses of indexes might be for filtering date ranges, or foreign key indexes when you want to get all rows of a given related dimension, depending on the cardinality of the relationship. For example, let's add two indexes:

Click here to view code image

```
CREATE UNIQUE INDEX [Sale Key] ON Fact.SaleBase ([Sale Key]);
CREATE INDEX [WWI Invoice ID] ON Fact.SaleBase ([WWI Invoice
ID]);
```

The [Sale Key] index is the surrogate key for the fact table, so it is unique, while the [WWI Invoice ID] is for the entire order, and the grain of the table is one row per invoice line item.

Now perform the query with the WHERE clause and you see the plan has changed to what you would desire for a single-row lookup, as you can see in Figure 1-32.

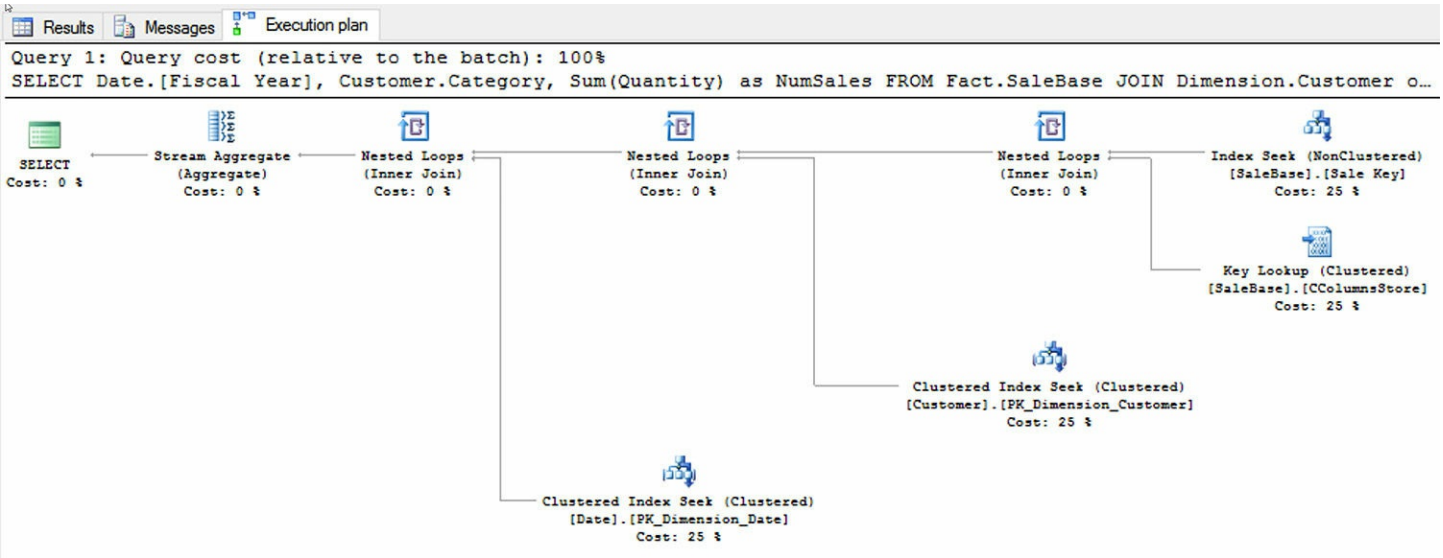


FIGURE 1-32 The plan with the query aggregating one row based on the [Sale Key] after adding a non-clustered rowstore index to the column

Figure 1-32 has the following output:

Click here to view code image

```
Table 'Date'. Scan count 0, logical reads 2, physical reads 0,
read-ahead reads 0, lob
logical reads 0, lob physical reads 0,

lob read-ahead reads 0.
```

```
Table 'Customer'. Scan count 0, logical reads 2, physical reads
0, read-ahead reads 0,
lob logical reads 0, lob physical reads 0,

lob read-ahead reads 0.

Table 'SaleBase'. Scan count 1, logical reads 3, physical reads
0, read-ahead reads 0,
lob logical reads 186, lob physical reads 0,

lob read-ahead reads 0.

Table 'SaleBase'. Segment reads 1, segment skipped 0.

CPU time = 0 ms,  elapsed time = 30 ms.
```

Now, while you never aggregate just a row at a time, other than perhaps in development, when you go to update a single row in the table during ETL, the operation of finding the rows is not cost prohibitive.

Implement columnstore index maintenance

In this section we look at what maintenance you need to do with your columnstore indexes as you load them in various ways. To do the index maintenance, use the ALTER INDEX command, either using the REORGANIZE or REBUILD settings. REORGANIZE basically starts the tuple mover immediately rather than running it in the background, slowly. REORGANIZE is, like the tuple mover running natively, an ONLINE operation. REBUILD on the other hand, is just like creating a new clean index, and compresses all of the data, but it is an offline process.

You need to decide whether to wait for the tuple mover to handle your structure, force the tuple mover to perform, or just rebuild your indexes, depending on the urgency of your needs. If this is a nightly-loaded data warehouse and you have the time, you want to just use REBUILD, but if it is a more active system, you want to check the configuration of the index and do a REORGANIZE. We look at a few examples in this section.

There are a few ways that data is loaded into a columnstore:

- **Bulk Load Into a Clustered Columnstore** Different from bulk loading data into a rowstore table, you can load bulk amounts of data into a clustered columnstore index by using an INSERT...SELECT ... FROM <TableName> WITH (TABLOCK) statement.
- **Other Batch Operations** Loading data where you don’t meet the requirements of the Bulk Load pattern.

For each of the examples, use a pared-down version of the Fact.Sale table named Fact.SaleLimited, as shown in Listing 1-18

LISTING 1-18 Fact table code to be used for this section

Click here to view code image

```
.CREATE TABLE [Fact].[SaleLimited](
       [City Key] [int] NOT NULL,
             [Customer Key] [int] NOT NULL,
       [Bill To Customer Key] [int] NOT NULL,
       [Stock Item Key] [int] NOT NULL,
       [Invoice Date Key] [date] NOT NULL,
       [Delivery Date Key] [date] NULL,
       [Salesperson Key] [int] NOT NULL,
       [WWI Invoice ID] [int] NOT NULL,
       [Description] [nvarchar](100) NOT NULL,
       [Package] [nvarchar](50) NOT NULL,
       [Quantity] [int] NOT NULL
);
```

Need More Review? ALTER INDEX review

Reviewing the more complex settings in the ALTER INDEX statement is a good thing to do. There are many settings that we are not able to review in this book: *https://msdn.microsoft.com/en-us/library/ms188388.aspx*.

Bulk loading data into a clustered columnstore

To start with, add a clustered columnstore index to the Fact.SaleLimited table:

Click here to view code image

```
CREATE CLUSTERED COLUMNSTORE INDEX [CColumnStore] ON [Fact].
[SaleLimited];
```

Next, load some data. The WITH (TABLOCK) allows this statement to run in parallel (for more information, the following blog has more details: *https://blogs.msdn.microsoft.com/sqlcat/2016/07/21/real-world-parallel-insert-what-else-you-need-to-know/)*, so you could end up with more or less deltastores on your system:

Click here to view code image

```
INSERT INTO [Fact].[SaleLimited] WITH (TABLOCK)
     ([City Key], [Customer Key],  [Bill To Customer Key],
[Stock Item Key],
      [Invoice Date Key], [Delivery Date Key],[Salesperson
Key],
      [WWI Invoice ID], [Description], [Package], [Quantity])
```

```
SELECT TOP (100000) [City Key], [Customer Key],  [Bill To
Customer Key],
          [Stock Item Key], [Invoice Date Key], [Delivery Date
Key],[Salesperson Key],
          [WWI Invoice ID], [Description], [Package],
[Quantity]
FROM Fact.Sale
GO 3 --run this statement 3 times
```

Next, go and look at the information from the DMV sys.dm_db_column_store_row_group_physical_stats, which gives you information about the physical characteristics of the rowgroups in your columnstore index. The query in Listing 1-19 is used throughout these maintenance sections to view the physical state of the columnstore indexes.

LISTING 1-19 Query on sys.dm_db_column_store_row_group_physical_stats to see state of columnstore indexes

Click here to view code image

```
SELECT state_desc, total_rows, deleted_rows,
       transition_to_compressed_state_desc as transition
FROM sys.dm_db_column_store_row_group_physical_stats
WHERE object_id  = OBJECT_ID('Fact.SaleLimited');
```

The output of this query, right after execution is:

Click here to view code image

```
state_desc      total_rows deleted_rows transition
--------------- ---------- ------------ ----------------------
-------
OPEN            150000     0            NULL
OPEN            150000     0            NULL
```

Two deltastore rowgroups were created, since there are two processors in my VM, and the bulk insert performs in parallel. If the VM had 8 processors, there could have been as many as 8 groups created. Since the total_rows column value is less than the 1048576 rows that is optimum, executing simple ALTER INDEX REORGANIZE does not change anything. The transition column tells you what triggered the row group to transition to a compressed state. There are other good bits of information not shown here, like the trim_reason that tells you why less than the expected maximum number of rows are in the rowgroup.

However, if you are not going to be adding any additional rows, you can force the tuple mover to compress these rowgroups by executing:

Click here to view code image

```
ALTER INDEX CColumnStore ON Fact.SaleLimited REORGANIZE
                                    WITH (COMPRESS_ALL_ROW_GROUPS =
ON);
```

Then you see:

Click here to view code image

```
state_desc      total_rows deleted_rows transition
--------------- ---------- ------------ ----------------------
-------
COMPRESSED      150000     0            REORG_FORCED
COMPRESSED      150000     0            REORG_FORCED
TOMBSTONE       150000     0            NULL
TOMBSTONE       150000     0            NULL
```

The two deltastore rowgroups have been compressed, and have then been tombstoned. When and whether you want to do these tasks largely depends on the type of table. For an OLTP table, where data is being loaded constantly, it may not be advantageous to reorganize the table, particularly if you are rapidly reaching the million row point. Of course, it is possible that you want to run the reorganize prior to a large reporting task. For a data warehouse table that is loaded periodically, you want to either rebuild or reorganize, depending on time allotted, and how many UPDATE and DELETE operations have occurred. Once you have compressed filegroups, things happen automatically.

Forcing the tuple mover to start by running REORGANIZE has the process combine the two compressed row groups, as the larger the number of rows in the rowgroup, up to the maximum, is better.

Click here to view code image

```
ALTER INDEX CColumnStore ON Fact.SaleLimited REORGANIZE;
```

Now check the structure of the columnstore index:

Click here to view code image

```
state_desc      total_rows deleted_rows transition
--------------- ---------- ------------ ----------------------
-------
COMPRESSED      300000     0            MERGE
TOMBSTONE       150000     0            NULL
TOMBSTONE       150000     0            NULL
```

Note, if you directly bulk insert 102400 or more rows, the data goes directly into compressed rowgroups as rows are being added. To show what happens when you bulk load at least the minimum number of rows, perform:

Click here to view code image

```
TRUNCATE TABLE Fact.SaleLimited;
GO
INSERT INTO [Fact].[SaleLimited] WITH (TABLOCK)
     ([City Key], [Customer Key],  [Bill To Customer Key],
[Stock Item Key],
      [Invoice Date Key], [Delivery Date Key],[Salesperson
Key],
      [WWI Invoice ID], [Description], [Package], [Quantity])
SELECT TOP (102400) [City Key], [Customer Key],  [Bill To
Customer Key],
          [Stock Item Key], [Invoice Date Key], [Delivery Date
Key],[Salesperson Key],
          [WWI Invoice ID], [Description], [Package],
[Quantity]
FROM Fact.Sale
OPTION (MAXDOP 1); --not in parallel
GO 3
```

You now see three compressed rowgroups in the output of the Listing 1-19 query. This is better for actually using the row groups immediately, but not as efficient as having them all in the same rowgroup:

Click here to view code image

```
state_desc      total_rows deleted_rows transition
--------------- ---------- ------------ ----------------------
-------
COMPRESSED      102400     0            BULKLOAD
COMPRESSED      102400     0            BULKLOAD
COMPRESSED      102400     0            BULKLOAD
```

Perform ALTER INDEX REORGANIZE, and these three rowgroups are combined into just 1.

Click here to view code image

```
state_desc      total_rows deleted_rows transition
--------------- ---------- ------------ ----------------------
-------
COMPRESSED      307200     0            MERGE
TOMBSTONE       102400     0            NULL
TOMBSTONE       102400     0            NULL
TOMBSTONE       102400     0            NULL
```

Using ALTER INDEX...REBUILD skips directly to having all of the rowgroups compressed in the best fashion possible, much like dropping and recreating the index, but it is an offline process just like initially recreating the index. This is true, even a minimal number of rows, as in this example:

Click here to view code image

```
TRUNCATE TABLE Fact.SaleLimited;
INSERT INTO [Fact].[SaleLimited] WITH (TABLOCK)
([City Key],     [Customer Key],
                 [Bill To Customer Key], [Stock Item Key],
                 [Invoice Date Key], [Delivery Date Key],
[Salesperson Key],
                 [WWI Invoice ID], [Description], [Package],
[Quantity])

SELECT TOP (5000) [City Key],     [Customer Key],
                 [Bill To Customer Key], [Stock Item Key],
                 [Invoice Date Key], [Delivery Date Key],
[Salesperson Key],
                 [WWI Invoice ID], [Description], [Package],
[Quantity]

FROM Fact.Sale;
```

Then rebuild the index:

Click here to view code image

```
ALTER INDEX [CColumnStore] ON [Fact].[SaleLimited] REBUILD;
```

And there is a compressed rowgroup with only 5000 rows:

Click here to view code image

```
state_desc      total_rows deleted_rows transition
--------------- ---------- ------------ ----------------------
-------
COMPRESSED      5000       0            INDEX_BUILD
```

Non-bulk operations on a columnstore

For any columnstore index, when you load data and never reach the 1048576 rows to get the tuple mover to compress the data, you can do the exact same tasks as we looked at in the previous section. The following code demonstrates how the other, non-bulk operations show up, and what maintenance steps that one can take to clear things up. Using the same table we had in the previous section, load 100000 rows, and use ALTER INDEX REBUILD to set the index up in pristine shape.

Click here to view code image

```
TRUNCATE TABLE Fact.SaleLimited;
INSERT INTO [Fact].[SaleLimited]
([City Key],     [Customer Key],
                 [Bill To Customer Key], [Stock Item Key],
```

```
                    [Invoice Date Key], [Delivery Date Key],
[Salesperson Key],
                    [WWI Invoice ID], [Description], [Package],
[Quantity])

SELECT TOP (100000 ) [City Key],      [Customer Key],
                     [Bill To Customer Key], [Stock Item Key],
                     [Invoice Date Key], [Delivery Date Key],
[Salesperson Key],
                     [WWI Invoice ID], [Description], [Package],
[Quantity]

FROM Fact.Sale;
ALTER INDEX [CColumnStore] ON [Fact].[SaleLimited] REBUILD;
```

Now the structure looks like:

Click here to view code image

```
state_desc      total_rows deleted_rows transition
--------------- ---------- ------------ ----------------------
-------
COMPRESSED      100000     0            INDEX_BUILD
```

Then delete some data:

Click here to view code image

```
DELETE FROM Fact.SaleLimited
WHERE  [Customer Key] = 21;
```

You now see what is sometimes thought of as fragmentation showing up. There are still 100000 rows in the rowgroup, but 135 rows are marked as deleted. As this number grows, the rowgroup becomes less and less useful:

Click here to view code image

```
state_desc      total_rows deleted_rows transition
--------------- ---------- ------------ ----------------------
-------
COMPRESSED      100000     135          INDEX_BUILD
```

Next, update some data:

```
UPDATE Fact.SaleLimited
SET    [Customer Key] = 35
WHERE  [Customer Key] = 22;
```

Looking at the structure, you can now see that we have more deleted rows, and a new delta rowgroup for the inserted versions of the updated rows.

Click here to view code image

```
state_desc      total_rows deleted_rows transition
--------------- ---------- ------------ ----------------------
-------
OPEN            98         0            NULL
COMPRESSED      100000     233          INDEX_BUILD
```

From here, there are three choices. Wait for the tuple mover to deal with this situation, though maybe not if more data isn't loaded. REBUILD, or REORGANIZE using the COMPRESS_ALL_ROW_GROUPS option as done in the previous section.

The steps done here are the same for non-clustered and clustered columnstore indexes, but the non-clustered version does not update the deleted_rows column, whereas the clustered does. Columnstore indexes are wonderful for many things, but they are definitely tuned for large quantities of data, particularly when the automated processes work after more than a million rows are inserted. This, plus the fact that they can get fragmented from just simple INSERT, UPDATE, and DELETE statements means you certainly want to keep up with how the load from your processes is affecting the rowgroups.

Need More Review? Columnstore index maintenance resources

There are a lot of resources about maintaining columnstore indexes, and a few of the more useful ones that are definitely worth reviewing are:

- **Columnstore Indexes Defragmentation** *https://msdn.microsoft.com/en-us/library/dn935013.aspx*. Details on what fragments columnstore indexes, and links to Sunil Agarwal's blogs on the topics as well.
- **Part 36 of Niko Neugebauer's blog series on clustered columnstore indexes** *http://www.nikoport.com/2014/07/29/clustered-columnstore-indexes-part-36-maintenance-solutions-for-columnstore/*. Of course, the other many sections of his series on columnstore indexes are useful reading as well.
- **An excellent blog on the Tuple Mover on the Rusanu Consulting Blog, mentioned by Sunil Agarwal** *http://rusanu.com/2013/12/02/sql-server-clustered-columnstore-tuple-mover/*.
- **Columnstore Data Loading** *https://msdn.microsoft.com/en-us/library/dn935008.aspx* Covers how data is loaded into the columnstore index, and how this affects the need to maintain your indexes.

Chapter summary

- Designing your database objects starts with requirements and understanding them. On the exam, if you are presented with a set of requirements to match to a table set, make sure you comprehend and match requirements precisely to the objects given.
- Understanding what normalization means is essential to the process of matching the

needs of the user to the needs of the query optimizer and processor. The normal forms are very much about eliminated duplication in your structures to enhance data integrity,

- SQL Server provides you as a user with a tremendous number of data types, not all of which you need for any given design. Match the data type you need to the requirements the user provides or, if no prescribed size is provided, to a size that is large enough to handle any reasonable value, but not so large as to allow unnecessarily large values. For example: nvarchar(3) for a person's name, too small, nvarchar(max) far too large. nvarchar(50) is probably more than you ever need, but not so large that it is ridiculous.
- Dynamic Data Masking can be added to the declaration for a column in the table create statement to mask data from database principals that do not have the UNMASK privilege in the database. There are several functions you can use to mask data.
- The clustered index of a table is a very important choice, as the key columns are used in all other rowstore indexes. The greater percentage of your queries can use the clustered index for seek operations to answer queries the better. The best clustered indexes are small in size, never changing, and monotonically increasing. A small clustering key size reduces the size of all nonclustered indexes; never changing values eliminates updates to non-clustered indexes; and monotonically increasing lets data be inserted at the end of the physical structures.
- An essential tool for designing indexes is the query plan. Use the graphical versions from the UI, or one of the SET SHOWPLAN commands to get a textual version of the estimated plan, or SET STATISTICS PROFILE ON for a textual version of the actual plan. Adding indexes without any real knowledge of how they are used often ends up creating useless indexes (something that is touched on in Chapter 4.)
- Index key columns can total 900 bytes or less for a clustered index, and up to 1700 bytes for a non-clustered index.
- There is only one clustered index, which makes it very important to choose what to put in the clustered index wisely for two reasons:
  - The data pages of the table are ordered by the clustering key
  - All non-clustered indexes use the clustering key for their row locators
- Providing summarized/denormalized data to the client by DML based automated means can be achieved using an indexed view.
- Columnstore indexes are specifically built for analytic purposes and typically need to be coupled with rowstore indexes to allow searches on a single row, unless you extremely rarely need to fetch a single row.
- Clustered columnstore indexes change your table's storage and compress your data considerably, reducing the amount of IO needed to perform queries on very large data

sets. Nonclustered columnstore indexes can be added to a rowstore table to allow real-time analytics.

- In columnstore indexes, DELETE operations just mark compressed rows as deleted, and UPDATE operations mark the row to be updated deleted and perform and INSERT. Both INSERT operations add rows to a deltastore rowgroup. Unless your table is very active, you need to perform maintenance on your table using ALTER INDEX REORGANIZE or ALTER INDEX REBUILD to get the most out of your columnstore indexes, as the background tuple mover moves rows when there are 1048576 rows in the deltastore rowgroup.

Thought experiment

In this thought experiment, demonstrate your skills and knowledge of the topics covered in this chapter. You can find the answer to this thought experiment in the next section.

You are taking over the table design for the invoicing system for a toy shop. The requirements state "Wingtip Toys want to track their customer's basic information, the orders they have placed for one or more products, and how much they have paid for each product on the order. The price of a toy can change for every order."

In the database, you see the following tables, with accompanying definitions:

- **Customer** People and companies that have purchase toys from the shop
- **CustomerOwnedToys** Toys that we know a customer owns
- **InvoiceItem** The bill for the toys that a customer has purchased
- **Product** The list of products that Wingtip Toys sells.

What are the potential issues with the design, based solely on the table names?

The original designer had created a column in the Customer table named: nickname, allowing you to store a name a person would prefer to be called. For example, a person named William might prefer to be called Bill, and Mariusz Wolodzko could prefer to be called "Captain Awesome." The data type for this column is int, and you realize that int is not really a desirable data type for storing a text value. You have the following five different data types that you are deciding from:

1. varchar (100)
2. nvarchar(50)
3. nvarchar(Max)
4. char(120)
5. varbinary(100)

For each type, evaluate in terms of A. Ability to store the data B. Appropriateness of the data type to meet the specified requirements.

Once the database has been created, the following query is written:

Click here to view code image

```
SELECT StockItemID, StockItemName, SupplierID, ColorID,
UnitPackageID,
       OuterPackageID, Brand, Size
FROM   Examples.Product
WHERE  UnitPackageId = 9;
```

And the plan for this query is shown in Figure 1-33:

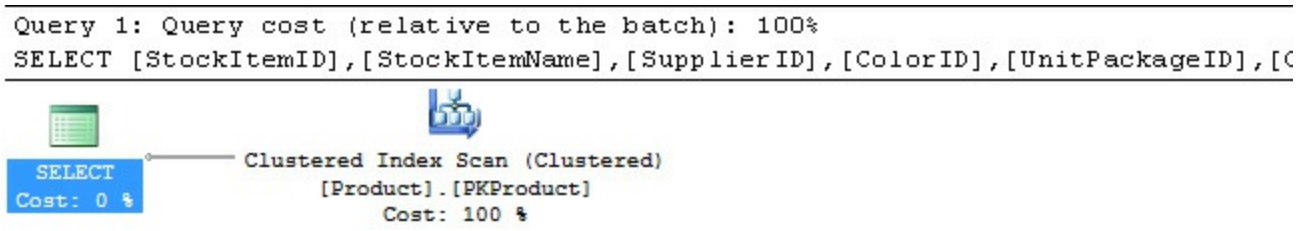


FIGURE 1-33 The plan for a simple query

From the information given, can you decide if adding either of the following indexes improves the query?

- CREATE INDEX UnitPackageId ON Examples.Product (UnitPackageId);
- CREATE INDEX UnitPackageIdPlus ON Examples.Product (UnitPackageId) INCLUDE (StockItemID, StockItemName, SupplierID, ColorID, OuterPackageID, Brand, Size);

Finally, in your review, you see the following table with over 5 million rows in it:

Click here to view code image

```
CREATE TABLE Sales.InvoiceItemFact
(
    InvoiceItemFactId int NOT NULL
          IDENTITY CONSTRAINT PKInvoiceItemFact PRIMARY KEY,
    ProductDimId int NOT NULL,
    CustomerDimId int NOT NULL,
    DateDimId int NOT NULL,
    SalesAmount money NOT NULL,
    SalesQuantity decimal(10,4)

);
```

There is also a table named Sales.ProductDim and Sales.CustomerDim, that contain data about the customer and product associated with the sale. Do all of the indexes work with the existing PRIMARY KEY constraint's index? For queries that aggregate a lot of data in the table, do the indexes help?

- CREATE INDEX ProductDimId ON Sales.InvoiceItemFact (ProductDimId);
- CREATE NONCLUSTERED COLUMNSTORE INDEX NCColumnstore ON

Sales.InvoiceItemFact (InvoiceItemFactId, ProductDimId, CustomerDimId, DateDimId, SalesAmount, SalesQuantity);

- CREATE CLUSTERED COLUMNSTORE INDEX CColumnstore ON Sales.InvoiceItemFact;

Thought experiment answer

This section contains the solution to the thought experiment. Each answer explains why the answer choice is correct. Users provide an email address when they sign up. However, sometimes the same person creates multiple accounts with the same email address, causing issues with the validity of a research results set.

There are several concerns in the design as given, both in terms of meeting the requirements given and basic normalization considerations. In terms of requirements, there is a table named CustomerOwnedToys, specifying a feature that was not requested by the customer. The normalization problem comes with just having an InvoiceItem table without a table named Invoice. This causes the columns to need to repeat information about the order that was placed. There was one answer included to possibly throw you off. "The price of a toy can change for every order" seems to indicate that we need a ProductPrice table. However, the requirements strictly ask "how much they have paid for each product on the order".

A column for a person's nickname is a typical need, and clearly int is not the proper solution. Let's look at each type mentioned, in terms of A. Ability to store the data, and B. Appropriateness of the data type to meet the specified requirements.

1. varchar (100)
   - **A.** Since this is an ASCII type, it is acceptable for storing simple character data. However, real names contain special characters. It is best to use a Unicode type for names
   - **B.** 100 characters of string data is likely too long of a string for name data, the width of text on this page is approximately 80 characters wide. If you are going to allow 100 characters in a column, usage should never cut it off. Since it is variably sized, it does not waste space, making it efficient enough.
2. nvarchar(50)
   - **A.** This is a Unicode type, and as such it should handle any text.
   - **B.** 50 characters is a typical length that people set names to be. Since it is variably sized, it does not waste space, making it efficient enough.
3. nvarchar(Max)
   - **A.** This is a Unicode type, and as such it should handle any text.
   - **B.** Much like other larger strings, this is too large of a string for a name at over 1 million characters. Some designers simply use nvarchar(max) because it is easy,

and since it is variably sized, technically no less efficient than any other nvarchar type. However, this is generally not the best practice that is desirable.

4. char(120)

   A. Since this is an ASCII type, it is acceptable for storing simple character data. However, real names contain special characters. It is best to use a Unicode type for names.

   B. 120 characters is too long, and in this case, since it is not variable sized, space is typically wasted when storing a short string. Even if using a compression setting (not covered in this exam), the string appears padded in all uses, which is not optimal.

5. varbinary(100)

   A. If you assumed that this would not work, you were incorrect. While it is definitely not the most efficient way to store a string, the following CAST expression results in a Unicode string of N'Bill': CAST (0x420069006C006C00 as nvarchar(100)).

   B. Clearly this is not optimum way of storing a string. However, this is something that could have been done in very old systems to give binary comparisons. Using a binary collation provides the same properties in modern versions of SQL Server.

Given the query we have been provided on the Examples.Product table, without any data to look at, it is not possible to be sure that either index that we are provided with gives any benefit (for example, what if the table had 1 row? The size of the line from the Clustered Index Scan to the SELECT operator is very thin. It also isn't stated if an index already exists on the UnitPackage column.) However, so this isn't just a trick question, let's assume that there is data in the table, there is not an index on UnitPackageId, and some data was returned from the query.

- **CREATE INDEX UnitPackageId ON Examples.Product (UnitPackageId);** This index has the least likelihood between the two to provide benefit. For it to be useful, there needs to be a lot of rows that don't match, and just a few that do, based on the statistics of the table. Because of the necessity of a Key Lookup operator to fetch additional columns that are not in the index, the value of this index has to be great to outweigh the costs.
- **CREATE INDEX UnitPackageIdPlus ON Examples.Product (UnitPackageId) INCLUDE (StockItemID, StockItemName, SupplierID, ColorID, OuterPackageID, Brand, Size);** There is no guarantee that this index tremendously improves the query, since we are not sure if there are columns in the table that are not included but the index, but since the index key matches the WHERE clause, and the INCLUDE columns match the SELECT clause, it can almost certainly be of some value to the query.

Finally, there were a few indexes that were to be applied to a fact table in a dimensional

design.

All indexes listed work with the existing PRIMARY KEY constraint index.

- **CREATE INDEX ProductDimId ON Sales.InvoiceItemFact (ProductDimId)**; This index is generally only be useful for queries only return a ProductDimId, such as counting the number of orders per ProductDimId. Generally, this is not a great, general-purpose index.
- **CREATE NONCLUSTERED COLUMNSTORE INDEX NCColumnstore ON Sales.InvoiceItemFact (InvoiceItemFactId, ProductDimId, CustomerDimId, DateDimId, SalesAmount, SalesQuantity);** Almost all queries that do analytical queries on the Sales.InvoiceItemFact table can benefit from this index. However, the non-clustered columnstore index is generally more appropriate for OLTP tables, where you want a minimal overhead for the OLTP users, and no change to the base table structures.
- **CREATE CLUSTERED COLUMNSTORE INDEX CColumnstore ON Sales.InvoiceItemFact;** This is the best choice of index from the list. It compresses the base data in the table to make the IO impact the smallest for all queries. It works nicely with the PRIMARY KEY constraint index to allow singleton updates/seeks as needed for ETL and simple queries also.

Chapter 2. Implement programmability objects

In the previous chapter, we reviewed the basic data structure of a SQL Server database. First, we designed the structures of the database from requirements, and then we built a set of tables and views to access the data, with indexes to make the queries using these objects perform well. In this chapter, we further enhance this database by using more of the tools that SQL Server provides to enhance data integrity through constraints and Transact-SQL coded objects.

Skill 2.1 starts with constraints, a topic we brushed upon in Chapter 1, but we dive deeper into their use and configuration. Constraints help shape the data that can be placed in your tables in a few ways, such as keeping data unique, limiting the domain of columns to certain formats or lengths that can't be done with data type alone, and enforcing foreign key relationships. Using constraints, you take databases from simple data storage, to intelligent data filters that eliminate most data issues with very little performance impact.

In Skill 2.2 we cover stored procedures, which allow you as a programmer to bundle the execution of code together in a way that is generally much more convenient to use than ad-hoc Transact-SQL statements from the middle layer of an application. Almost any code can be performed from a stored procedure, and we cover several of the most useful scenarios.

In Skill 2.3, the final skill section of this chapter, there are two types of objects we deal with: triggers and User-Defined Functions (UDFs). TRIGGER objects are used to react to certain types of actions, such as a DML operation like an INSERT, a DDL operation like ALTER INDEX, CREATE TABLE, etc., or a user logging into the server. The most common use of triggers is to use a DML trigger to extend data integrity where constraints are not sufficient.

UDFs allow you to create modules of code that can be used as the building blocks of SQL statements. While Transact-SQL is not generally great at producing very modular code, UDFs allow you to build code that works much like SQL Server's system functions where it makes sense.

> **Note Transact-SQL solutions**
>
> This chapter focuses primarily on on-disk tables and interpreted Transact-SQL solutions exclusively. Skill 3.4 highlights the differences between these solutions and the memory-optimized tables and natively-compiled modules.

Skills in this chapter:

- Ensure data integrity with constraints
- Create stored procedures
- Create triggers and user-defined functions

Skill 2.1 Ensure data integrity with constraints

In Chapter 1, the first skill that we explored was designing a database. In that exercise, we designed a database that met some basic requirements. Many of the data integrity requirements for a database are covered by the table design and the physical implementation, but not all of them. In this skill section, we look at the declarative tools that are available to further constrain data to meet a set of requirements.

This section covers how to:

- Define table and foreign-key constraints to enforce business rules
- Write Transact-SQL statements to add constraints to tables
- Identify results of Data Manipulation Language (DML) statements given existing tables and constraints
- Identify proper usage of PRIMARY KEY constraints

Define table and foreign-key constraints to enforce business rules

PRIMARY KEY constraints are almost always used by even novice database implementers, but for the exam (and for implementing a proper database), the other types of constraints that are available are extremely important as well. Constraints can either be for a single column (referred to as column constraints), or for multiple columns (referred to as table constraints.) In this section, we review the following constraint types that you should be familiar with:

- **DEFAULT** Used to provide a value for a column when no value is provided by the user.
- **UNIQUE** Used to implement any uniqueness criteria (alternate/candidate keys) that are not chosen as the primary key.
- **CHECK** Used to apply a simple predicate check to the values in an INSERT or UPDATE operation.
- **FOREIGN KEY** Used to enforce foreign key relationships between tables, so that referenced data is always in existence for rows that reference it.

In addition to reviewing each of these types of constraint individually, we also examine a section on limiting a column value to a set of values, which makes use of a few of these items simultaneously.

Using DEFAULT constraints to guide the user's input

DEFAULT constraints, at first look, don't seem like they have much value for enforcing business rules, and that is somewhat true. However, they are useful to give the user an idea of what value to put in a column. For example, say you have a column in a table that is

called DisabledFlag, with a domain of 1, if what the row represents has been disabled, and 0 when not. More than likely, the typical value of this column is 0. So in the table definition, you might specify:

Click here to view code image

```
DisabledFlag bit NOT NULL CONSTRAINT DFTLTableName_DisabledFlag
DEFAULT (0);
```

Now, if a user doesn't specify this value in the INSERT, it is automatically 0. Applications can access the metadata of the default value as well, so it can be useful that way as well (this can be accessed in sys.default_constraints). There are a few system uses of DEFAULT constraints as well that are commonly used. One is to make it easier to set columns that are used to denote when a row was modified, created, etc. such as RowLastModifiedTime (RowCreatedTime, RowCreatedByLogin, and others). For example, consider the following table, with just a simple integer primary key, and a column that is used to capture the last time the row was modified:

Click here to view code image

```
CREATE SCHEMA Examples;
GO
CREATE TABLE Examples.Widget
(
    WidgetId    int CONSTRAINT PKWidget PRIMARY KEY,
    RowLastModifiedTime datetime2(0) NOT NULL
);

Add the DEFAULT constraint as the default value for the column:

ALTER TABLE Examples.Widget
  ADD CONSTRAINT DFLTWidget_RowLastModifiedTime
         DEFAULT (SYSDATETIME()) FOR RowLastModifiedTime;
```

So if you insert a row, you can do one of two things. Either don't include the column in the INSERT statement, or use the DEFAULT keyword to have the value default itself, as in the following two statements:

Click here to view code image

```
INSERT INTO Examples.Widget(WidgetId)
VALUES (1),(2);
INSERT INTO Examples.Widget(WidgetId, RowLastModifiedTime)
VALUES (3,DEFAULT), (4,DEFAULT);
```

Checking the data that has been inserted:

```
SELECT *
FROM   Examples.Widget;
```

The values are all the same, as the statements were executed within the same second:

Click here to view code image

```
WidgetId    RowLastModifiedTime
----------- --------------------------
1           2016-09-14 18:08:28
2           2016-09-14 18:08:28
3           2016-09-14 18:08:28
4           2016-09-14 18:08:28
```

You can also use the DEFAULT keyword on an UPDATE operation. The following query would change every row's RowLastModifiedTime to the default value, which is the current time:

Click here to view code image

```
UPDATE Examples.Widget
SET RowLastModifiedTime = DEFAULT;
```

DEFAULT constraints are also useful for adding a new NOT NULL column to a table that already has data in it. As the column is being added to the table, it uses the DEFAULT constraints value. If you are adding a DEFAULT to an existing column, use the following syntax:

Click here to view code image

```
ALTER TABLE Examples.Widget
  ADD EnabledFlag BIT NOT NULL
      CONSTRAINT DFLTWidget_EnabledFlag DEFAULT (1);
```

Note NULL columns

If the column is defined as NULL when creating it, all of the values are NULL in the column when you create the column even if you attach a DEFAULT constraint. When you are creating a NULL column and want to default all of the values in the new column regardless, use WITH VALUES after the DEFAULT specification DEFAULT (value) WITH VALUES.

One last interesting thing you should know about using DEFAULT constraints is that if every column either has the IDENTITY property or has a DEFAULT constraint, you can use DEFAULT VALUES to skip the entire VALUES clause. For example, consider the following table:

Click here to view code image

```
CREATE TABLE Examples.AllDefaulted
(
    AllDefaultedId int IDENTITY(1,1) NOT NULL,
```

```
    RowCreatedTime datetime2(0) NOT NULL
        CONSTRAINT DFLTAllDefaulted_RowCreatedTime DEFAULT
(SYSDATETIME()),
    RowModifiedTime datetime2(0) NOT NULL
        CONSTRAINT DFLTAllDefaulted_RowModifiedTime DEFAULT
(SYSDATETIME())
);
```

Now you can create a new row with all default values, by using DEFAULT VALUES:

Click here to view code image

```
INSERT INTO Examples.AllDefaulted
DEFAULT VALUES;
```

You can specify any of the columns in the table in the INSERT INTO clause that have a DEFAULT constraint:

Click here to view code image

```
INSERT INTO Examples.AllDefaulted(RowModifiedTime,
RowCreatedTime)
DEFAULT VALUES;
INSERT INTO Examples.AllDefaulted(RowCreatedTime)
DEFAULT VALUES;
```

And the values are defaulted:

Click here to view code image

```
SELECT *
FROM   Examples.AllDefaulted;
```

This returns:

Click here to view code image

```
AllDefaultedId RowCreatedTime              RowModifiedTime
-------------- --------------------------- -------------------
-------
1              2016-09-14 18:19:30         2016-09-14 18:19:30
2              2016-09-14 18:19:30         2016-09-14 18:19:30
3              2016-09-14 18:19:30         2016-09-14 18:19:30
```

As we mentioned previously, this does not work with the column with the IDENTITY property set. So if you were to include the column in the INSERT column list, you will get an error:

Click here to view code image

```
INSERT INTO Examples.AllDefaulted(AllDefaultedId)
DEFAULT VALUES;
```

This gets you an error message:

Click here to view code image

```
Msg 339, Level 16, State 1, Line 69
DEFAULT or NULL are not allowed as explicit identity values.
```

This is because not including the column in the INSERT list is telling the query processor to use DEFAULT constraint values, not to use the IDENTITY property.

Using UNIQUE constraints to enforce secondary uniqueness criteria

A particularly important constraint to use when using surrogate keys for your primary keys is the UNIQUE constraint. We discuss choosing a PRIMARY KEY later in the chapter, but the purpose of the UNIQUE constraint is pretty straightforward: protect the uniqueness characteristics of column sets that need to be unique but were not chosen as the PRIMARY KEY.

Consider the following table that has two key columns, the GadgetId, and the GadgetCode. Say that GadgetId has been chosen as the PRIMARY KEY:

Click here to view code image

```
CREATE TABLE Examples.Gadget
(
    GadgetId    int IDENTITY(1,1) NOT NULL CONSTRAINT PKGadget
PRIMARY KEY,
    GadgetCode  varchar(10) NOT NULL
);
```

Now the following set of data is inserted:

Click here to view code image

```
INSERT INTO Examples.Gadget(GadgetCode)
VALUES ('Gadget'), ('Gadget'), ('Gadget');
```

The data in the table now looks like the following:

```
GadgetId    GadgetCode
----------- ----------
1           Gadget
2           Gadget
3           Gadget
```

It is not possible to tell one row from another except using a value that was system-generated, so we need to add a constraint to the table to make sure that this cannot happen. The UNIQUE constraint works very much like a PRIMARY KEY constraint, in that it enforces uniqueness and is implemented with an UNIQUE INDEX. There are a few subtle differences however:

- The index that is created to back the constraint is nonclustered by default.
- The columns of the key allow NULL values (NULL values are treated as distinct values, as was covered in Chapter 1, Skill 2.1, Indexing during the database design phase, where we first mentioned uniqueness constraints).

On the GadgetCode column of the Examples.Gadget table, create a UNIQUE constraint, after deleting the logically duplicated data:

Click here to view code image

```
DELETE FROM Examples.Gadget WHERE GadgetId in (2,3);

ALTER TABLE Examples.Gadget
   ADD CONSTRAINT AKGadget UNIQUE (GadgetCode);
```

Now, an attempt to insert a row with the duplicated tag value of G001:

Click here to view code image

```
INSERT INTO Equipment.Tag(Tag, TagCompanyId)
VALUES ('G001',1);
```

Instead of creating duplicated data in the column, this results in the following error:

Click here to view code image

```
Msg 2627, Level 14, State 1, Line 100
Violation of UNIQUE KEY constraint 'AKGadget'. Cannot insert
duplicate key in object
'Examples.Gadget'. The duplicate key value is (Gadget).
```

Back in Chapter 1, when talking about indexes, we previously covered the concerns with having NULL columns in your UNIQUE constraints. UNIQUE (and PRIMARY KEY) constraints are objects that have properties of data integrity protection, which this skill section is about, as well as indexes.

> **Need More Review? Creating UNIQUE contstraints**
>
> See the following article on MSDN for more details about creating UNIQUE constraints: *https://msdn.microsoft.com/en-us/library/ms190024.aspx*.

Using CHECK constraints to limit data input

The CHECK constraint is used to apply an expression predicate to data as it is inserted or updated. When evaluating the predicate of a CHECK constraint, the expression must evaluate to FALSE before the new or changed data is rejected. If a column allows NULL, and the expression does not explicitly reject NULL values, then if you need the constraint to fail on any condition, you must explicitly check for NULL.

Typical uses of CHECK constraints are to validate the format of a piece of data, limit the domain of data stricter than a data type, ensure data is in a valid range, and to coordinate multiple values make sense together (the last section of this skill review uses CHECK constraints as one method of choosing an explicit domain of values). The constraint can use a simple expression, and even use a user-defined function that accesses other tables, though that is not a typical use.

Using our sample database, there are several places where we need to limit the data that can be put into the tables. We look at:

- **Limiting data more than a data type** For example, the int data type is arguably the most common data type, but usually the desired range of a columns' value is not between approximately -2 billion to 2 billion. A CHECK constrain can limit the data in a column to a desired range.
- **Enforcing a format for data in a column** Some values, usually character data, needs to meet a predefined format. For example, an American Social Security Number is formatted NNN-NN-NNNN where N is a whole number.
- **Coordinate multiple values together** In some cases, multiple columns need to make logical sense together. For example, a composite foreign key reference that allows NULL values.

While the concept of a CHECK constraint is very simple, in practice there is one major thing to remember: in building a database (and possibly answering an exam question), if the requirement says "always" or "must" (as in "the maximum price of a widget must always be less than or equal to 100") this is a candidate for a constraint. If the requirement is less strict, (as in "the typical maximum price of a widget is 100"), a constraint cannot be used. This particular sort of constraint is more tailored to a user interface message box that asks: "Are you sure that they paid 200 for that widget?"

> **Need More Review? CHECK Constraints**
>
> For more information about CHECK constraints than we can cover, check out this article on MSDN about UNIQUE Constraints and CHECK constraints: *https://msdn.microsoft.com/en-us/library/ms187550.aspx#Check*.

Limiting data more than a data type

When creating initial database objects, a goal from Chapter 1, Skill 1.1 was to choose the best data type possible. If, for example, you need a data type that holds values between 1 and 10, you almost certainly choose a tinyint data type. The tinyint data type has a domain of 0 to 255, which is the data type with the best performance characteristics that is the smallest in range. You can use a decimal(2,0) to get to a domain of 0-99, but any integer type is better than a type that is implemented in the software of SQL Server rather than using the hardware as an integer would. In order to limit the values to between 1 and 10,

we will use a CHECK constraint.

For example, let's say you have a table that captures the cost of a product in a grocery store. You can use the smallmoney data type, but the smallmoney data type has a range of -214,748.3648 to 214,748.3647. There are concerns at the top and the bottom of the range. First, a product would not cost a negative amount, so the bottom limit should be at least 0. At the top you don't want to accidentally charge 200 thousand for a can of corn. For this example, we limit the cost to a range of greater than 0 to 999,9999.

Click here to view code image

```
CREATE TABLE Examples.GroceryItem
(
   ItemCost smallmoney NULL,
   CONSTRAINT CHKGroceryItem_ItemCostRange
       CHECK (ItemCost > 0 AND ItemCost < 1000)
);
```

Note Checking a constraint

You can determine if a constraint is a table or column level constraint by checking the parent_column_id in the sys.check_constraints system catalog view. If it is NULL, then it is a table constraint.

Now, any attempt to put a value outside of the range in the predicate:

Click here to view code image

```
INSERT INTO Examples.GroceryItem
VALUES (3000.95);
```

This causes an error:

Click here to view code image

```
Msg 547, Level 16, State 0, Line 286
The INSERT statement conflicted with the CHECK constraint
"CHKGroceryItem_ItemCostRange". The conflict occurred in
database "ExamBook762Ch2",
table "Examples.GroceryItem", column 'ItemCost'.
```

But values in the allowable range are accepted:

Click here to view code image

```
INSERT INTO Examples.GroceryItem
VALUES (100.95);
```

Finally, note that since the column allows NULL values, an INSERT with a NULL for the ItemCost is allowed, even though the predicate was: ItemCost > 0 and ItemCost < 1000.

Click here to view code image

```
INSERT INTO Examples.GroceryItem
VALUES (NULL);
```

If, for some reason, you want this column to reject NULL values even though it is declared NULL, you can add AND ItemCost IS NOT NULL to the predicate.

Enforcing a format for data in a column

Datatypes can be used to limit data to a maximum length, but they cannot limit data to a minimum length or a certain format (though XML and uniqueidentifier are examples where they have some formatting control). For example, it is a common desire to disallow a user from inputting only space characters for a value in a column, or to make sure that a corporate-standard-formatted value is input for a value.

As an example, consider the following table:

Click here to view code image

```
CREATE TABLE Examples.Message
(
    MessageTag  char(5) NOT NULL,
    Comment nvarchar(max) NULL
);
```

For these tables, we want to check the format of the two values. For the MessageTag, we want to make sure the format of the data is Alpha-NumberNumberNumber. For the Comment column, the requirement is to make sure that the value is either NULL, or a character string of 1 or more characters.

Click here to view code image

```
ALTER TABLE Examples.Message
   ADD CONSTRAINT CHKMessage_MessageTagFormat
      CHECK (MessageTag LIKE '[A-Z]-[0-9][0-9][0-9]');

ALTER TABLE Examples.Message
   ADD CONSTRAINT CHKMessage_CommentNotEmpty
       CHECK (LEN(Comment) > 0);
```

One of the primary difficulties regarding constraints (and really any of the declarative forms of data integrity checks we are reviewing) is that you only get one error, no matter how many errors are found. For example, say you break both rules in your statement:

Click here to view code image

```
INSERT INTO Examples.Message(MessageTag, Comment)
VALUES ('Bad','');
```

The only message you get back is for the MessageTag being poorly formatted (the order

of error checking is not guaranteed or controllable.):

Click here to view code image

```
Msg 547, Level 16, State 0, Line 312
The INSERT statement conflicted with the CHECK constraint
"CHKMessage_MessageTagFormat".
The conflict occurred in database "ExamBook762Ch2", table
"Examples.Message",
column 'MessageTag'.
```

Coordinate multiple values together

As one last example, consider a case where two column values can influence the legal value for another. For example, say you have a Customer table, and it has a set of status flags. Two of them are ForcedDisabledFlag, manually saying that the customer has been disabled, and a ForcedEnabledFlag, manually saying that the customer has been enabled, likely overriding the normal business rules in each case. Typically, there might be a few other columns for the user to explain why they are overriding the rules, but for simplicity, just these two columns are needed for the example.

The following table implements these two columns and a CHECK constraint that makes sure the offending scenario does not occur:

Click here to view code image

```
CREATE TABLE Examples.Customer
(
    ForcedDisabledFlag bit NOT NULL,
    ForcedEnabledFlag bit NOT NULL,
    CONSTRAINT CHKCustomer_ForcedStatusFlagCheck
      CHECK (NOT (ForcedDisabledFlag = 1 AND ForcedEnabledFlag
= 1))
);
```

Using FOREIGN KEY constraints to enforce relationships

FOREIGN KEY constraints are used to ensure that when you set up a foreign key link between tables (by placing the key value of one table in another table as a reference), the values remain in sync. They are generally quite simple to set up, though there are a number of options you can use to control what occurs when a reference exists, and when you are changing one side to not exist. In the next sections, we cover:

- Creating a simple FOREIGN KEY constraint on a table with data in it
- Cascading Operations
- Relating a table to itself to form a hierarchy
- FOREIGN KEY constraints relating to a UNIQUE constraint instead of a PRIMARY KEY constraint

Need More Review? FOREIGN KEY constraints

For more information about FOREIGN KEY constraints and their relationship to PRIMARY KEY constraints beyond what we can cover here, see the following article on MSDN: *https://msdn.microsoft.com/en-us/library/ms179610.aspx*.

Creating a simple FOREIGN KEY constraint on a table with data in it

Most FOREIGN KEY constraints that are implemented are of the straightforward variety. We need to make sure the data in one column in a table matches the data in the primary of another. In later sections, we cover some more breadth of configurations, but in this first section we keep it very simple.

For example, consider the following two tables, named after the common names for their position in the relationship (the Child table in a relationship references the Parent table).

Click here to view code image

```
CREATE TABLE Examples.Parent
(
    ParentId   int NOT NULL CONSTRAINT PKParent PRIMARY KEY
);
CREATE TABLE Examples.Child
(
    ChildId int NOT NULL CONSTRAINT PKChild PRIMARY KEY,
    ParentId int NULL
);
```

At this point, the user can put any value into the ParentId column of the Child table, which makes using the data complicated. To make sure that the data is always in sync, we can add the following constraint:

Click here to view code image

```
ALTER TABLE Examples.Child
     ADD CONSTRAINT FKChild_Ref_ExamplesParent
       FOREIGN KEY (ParentId) REFERENCES
Examples.Parent(ParentId);
```

In the declaration, you specify the column that references a given column in a table. While almost every FOREIGN KEY constraint references a PRIMARY KEY constraint, it can actually reference a UNIQUE constraint or even a UNIQUE index in the rare cases where that makes sense (more on that later in the section).

Now, after inserting a few rows into the Examples.Parent table:

Click here to view code image

```
INSERT INTO Examples.Parent(ParentId)
VALUES (1),(2),(3);
```

You are able to insert a row into Child where the ParentId does match:

Click here to view code image

```
INSERT INTO Examples.Child (ChildId, ParentId)
VALUES (1,1);
```

But if you try to use a ParentId that is not in the table:

Click here to view code image

```
INSERT INTO Examples.Child (ChildId, ParentId)
VALUES (2,100);
```

The following error is then thrown:

Click here to view code image

```
Msg 547, Level 16, State 0, Line 124
The INSERT statement conflicted with the FOREIGN KEY constraint
"FKChild_Ref_ExamplesParent". The conflict occurred in database
"ExamBook762Ch2",
table "Examples.Parent", column 'ParentId'.
```

Finally, note that the ParentId column in the Child table was created to allow NULL values. The referenced PRIMARY KEY constraint does not allow NULL values by definition, so this could never have a match. This brings up an important point about constraints. Much like CHECK constraints, they fail only when the comparison is FALSE, and any comparison to NULL return UNKNOWN. Hence the following INSERT statement works:

Click here to view code image

```
INSERT INTO Examples.Child (ChildId, ParentId)
VALUES (3,NULL);
```

So far, we have dealt with simple keys only, but PRIMARY KEY constraints can easily have a composite key. For required, NOT NULL child table references, this is not a concern. However, where the referencing table's columns do allow NULL values, something more complex occurs. Consider the following tables, and a single row in the table that the FOREIGN KEY constraint is referencing:

Click here to view code image

```
CREATE TABLE Examples.TwoPartKey
(
    KeyColumn1  int NOT NULL,
    KeyColumn2  int NOT NULL,
```

```
    CONSTRAINT PKTwoPartKey PRIMARY KEY (KeyColumn1,
KeyColumn2)
);

INSERT INTO Examples.TwoPartKey (KeyColumn1, KeyColumn2)

VALUES (1, 1);
CREATE TABLE Examples.TwoPartKeyReference
(
    KeyColumn1 int NULL,
    KeyColumn2 int NULL,
    CONSTRAINT FKTwoPartKeyReference_Ref_ExamplesTwoPartKey
        FOREIGN KEY (KeyColumn1, KeyColumn2)
            REFERENCES Examples.TwoPartKey (KeyColumn1,
KeyColumn2)
);
```

Here you put in a row with 1,1 for the Examples.TwoPartKeyReference table or NULL, NULL:

Click here to view code image

```
INSERT INTO Examples.TwoPartKeyReference (KeyColumn1,
KeyColumn2)
VALUES (1, 1), (NULL, NULL);
```

It is successful. If you try to put in 2,2, which is not in the referenced table:

Click here to view code image

```
INSERT INTO Examples.TwoPartKeyReference (KeyColumn1,
KeyColumn2)
VALUES (2, 2);
```

This does not work, as expected:

Click here to view code image

```
   Msg 547, Level 16, State 0, Line 157
The INSERT statement conflicted with the FOREIGN KEY constraint
"FKTwoPartKeyReference_
Ref_ExamplesTwoPartKey". The conflict occurred in database
"ExamBook762Ch2", table "Examples.TwoPartKey".
```

However, what about 6 million (a value most certainly not in the parent table) and NULL?

Click here to view code image

```
INSERT INTO Examples.TwoPartKeyReference (KeyColumn1,
KeyColumn2)
```

```
VALUES (6000000, NULL);
```

This actually works because the NULL is allowed by the column, and any column comparison that returns UNKNOWN (NULL) is accepted. To prevent this condition from occurring, you can use a CHECK constraint to make sure both columns are either NULL or NOT NULL. In this case, we could add the following CHECK constraint to correct this:

Click here to view code image

```
ALTER TABLE Alt.TwoPartKeyReference
      ADD CONSTRAINT CHKTwoPartKeyReference_FKNULLs
            CHECK ((KeyColumn1 IS NULL and KeyColumn2 IS NULL)
                   OR
                   (KeyColumn1 IS NOT NULL and KeyColumn2 IS
NOT NULL));
```

Now, the entire reference is NULL or NOT NULL, not one or the other, eliminating the concept that a KeyColumn1 or KeyColumn2 value might not exist in the referenced table.

Cascading Operations

We have seen already that a FOREIGN KEY constraint can be used to prevent rows being deleted from a table with referring data, or inserting or updating data into the referring table that doesn't match the referenced table. However, sometimes it is desirable to allow changes at the parent table to be reflected in the referencing child table. The following settings can be chosen when the row is deleted, or when the key columns in the parent are updated.

- **NO ACTION** Prevent any updates or deletions where the end result would leave the data invalid. This behaves as seen in the previous section, as this is the default action.
- **CASCADE** Repeat on the referencing table what occurs in the referenced. If the key column is changed, change it in the referencing table. If the row is deleted, remove it from the referencing table as well.
- **SET (NULL or DEFAULT)** In these cases, if the referenced row is deleted or the key value is changed, the referencing data is set to either NULL or to the value from a DEFAULT constraint, respectively.

The most common use of this feature is to cascade a DELETE operation to remove all related rows from one table to a related table that is, in essence, part of the referenced table. This is usually the case when one table logically owns the rows in the second table. For example, an invoice and invoice line item. You would never need an invoice line item without the invoice.

Click here to view code image

```
CREATE TABLE Examples.Invoice
```

```
(
    InvoiceId   int NOT NULL CONSTRAINT PKInvoice PRIMARY KEY
);
CREATE TABLE Examples.InvoiceLineItem
(
    InvoiceLineItemId int NOT NULL CONSTRAINT PKInvoiceLineItem
PRIMARY KEY,
    InvoiceLineNumber smallint NOT NULL,
    InvoiceId     int NOT NULL
       CONSTRAINT FKInvoiceLineItem_Ref_ExamplesInvoice
          REFERENCES Examples.Invoice(InvoiceId)
            ON DELETE CASCADE
            ON UPDATE NO ACTION,
    CONSTRAINT AKInvoiceLineItem UNIQUE (InvoiceId,
InvoiceLineNumber)
);
```

Now create a few rows of data in both tables:

Click here to view code image

```
INSERT INTO Examples.Invoice(InvoiceId)
VALUES (1),(2),(3);
INSERT INTO Examples.InvoiceLineItem(InvoiceLineItemId,
InvoiceId,InvoiceLineNumber)
VALUES (1,1,1),(2,1,2), (3,2,1);
```

View the data using the following query, which shows you the key of both tables (and the FULL OUTER JOIN insures that if we had a row in InvoiceLineItem without a referenced Invoice, it would still show up. This, however, is not possible, but if you were testing your code, this is the safest way to check the data.):

Click here to view code image

```
SELECT Invoice.InvoiceId, InvoiceLineItem.InvoiceLineItemId
FROM   Examples.Invoice
          FULL OUTER JOIN Examples.InvoiceLineItem
            ON Invoice.InvoiceId = InvoiceLineItem.InvoiceId;
```

This returns:

Click here to view code image

```
InvoiceId   InvoiceLineItemId
----------- -----------------
1           1
1           2
2           3
3           NULL
```

Now delete InvoiceId number 1:

```
DELETE Examples.Invoice
WHERE  InvoiceId = 1;
```

Repeat the query of the data, and you see that the Invoice and InvoiceLineItem rows have gone away.

Click here to view code image

```
InvoiceId   InvoiceLineItemId
----------- -----------------
2           3
3           NULL
```

When using surrogate keys for your primary key values (as we did with InvoiceId and InvoiceLineItemId), there are only a few scenarios for cascading update operations. Surrogate key values should never be updated. Sometimes this is technically possible, such as using a GUID for the key, but not needing to change the value is one of the main reasons why we use a surrogate key in the first place. However, if you choose to use natural keys for the primary key value, occasionally a value needs to be changed, either because of something like a company changing name, or perhaps because a misspelled value caught after the value was used in multiple places.

As an example, consider the following tables. The second table is an example of creating a FOREIGN KEY constraint in the table declaration, where the Example.Code table sets up a domain of code values, and Example.CodedItem simulates a row that needs that code (though the only column in the table is the Code column to keep things very simple).

Click here to view code image

```
CREATE TABLE Examples.Code
(
    Code    varchar(10) NOT NULL CONSTRAINT PKCode PRIMARY KEY
);
CREATE TABLE Examples.CodedItem
(
    Code    varchar(10) NOT NULL
         CONSTRAINT FKCodedItem_Ref_ExampleCode
            REFERENCES Examples.Code (Code)
                ON UPDATE CASCADE
);
```

Now, create a row in each table, with a misspelled code value of ‘Blacke’:

Click here to view code image

```
INSERT INTO Examples.Code (Code)
VALUES ('Blacke');
```

```
INSERT INTO Examples.CodedItem (Code)
VALUES ('Blacke');
```

Now, looking at the data, you can see that the data in both tables are spelled incorrectly:

Click here to view code image

```
SELECT Code.Code, CodedItem.Code AS CodedItemCode
FROM   Examples.Code
          FULL OUTER JOIN Examples.CodedItem
               ON Code.Code = CodedItem.Code;
```

This returns:

```
Code       CodedItemCode
---------- -------------
Blacke     Blacke
```

Now, update the Alt.Code row with the proper spelling of Black:

```
UPDATE Examples.Code
SET   Code = 'Black';
```

Check the data again, and see that both items say Black, as the UPDATE operation cascaded.

```
Code       CodedItemCode
---------- -------------
Black      Black
```

Relating a table to itself to form a hierarchy

In many databases, there is need to define a hierarchy of items. A common example is an employee-to-manager relationship, where everyone in a company except for one (the CEO or President) has a simple manager relationship. In the next example, we create a table named Examples.Employee that includes the relationship structure that one might create for an employee hierarchy. In the table, note that the ManagerId FOREIGN KEY constraint references the EmployeeId column in this same table:

Click here to view code image

```
CREATE TABLE Examples.Employee
(
         EmployeeId int NOT NULL CONSTRAINT PKEmployee PRIMARY
KEY,
         EmployeeNumber char(8) NOT NULL,
         ManagerId int NULL
               CONSTRAINT FKEmployee_Ref_ExamplesEmployee
                   REFERENCES Examples.Employee (EmployeeId);
```

```
);
```

Now you can add some data to the table, and add four rows, including the top level manager, and two persons that work for the top-level manager. Finally, one person works for one of those two employees:

Click here to view code image

```
INSERT INTO Examples.Employee(EmployeeId, EmployeeNumber,
ManagerId)
VALUES (1,'00000001',NULL), (2,'10000001',1),(3,'10000002',1),
(4,'20000001',3);
```

Now, check the contents of the table:

```
SELECT *
FROM   Examples.Employee;
```

By following the relationships in the data, you can see that EmployeeId = 1 is the main manager, and EmployeeId in (2,3) reports to number 1, and EmployeeId = 4 reports to EmployeeId = 3.

Click here to view code image

```
EmployeeId  EmployeeNumber ManagerId
----------- -------------- -----------
1           00000001       NULL
2           10000001       1
3           10000002       1
4           20000001       3
```

There's no need to go into any detail since this is not a Transact-SQL focused exam, but note that this sort of structure is typically queried via a recursive common table expression (CTE) such as the following:

Click here to view code image

```
WITH EmployeeHierarchy AS
(
    SELECT EmployeeID,  CAST(CONCAT('\',EmployeeId,'\') AS
varchar(1500)) AS Hierarchy
    FROM HumanResources.Employee
    WHERE ManagedByEmployeeId IS NULL
    UNION ALL
    SELECT Employee.EmployeeID,
CAST(CONCAT(Hierarchy,Employee.EmployeeId,'\')
                                                        AS
varchar(1500)) AS Hierarchy
    FROM HumanResources.Employee
      INNER JOIN EmployeeHierarchy
```

```
         ON Employee.ManagedByEmployeeId =
EmployeeHierarchy.EmployeeId
  )
SELECT *
FROM   EmployeeHierarchy;
```

This returns:

```
EmployeeID  Hierarchy
----------- ---------------
1           \1\
2           \1\2\
3           \1\3\
4           \1\3\4\
```

The Hierarchy column is a delimited display of the path from the row with NULL for ManagerId to the EmployeeId in the row.

> **Need More Review Common Table Expressions**
>
> If you would like to read more about CTEs, the following article by Robert Shelton on Simple-Talk gives a very easy-to-follow description of using them, including recursive CTEs: *https://www.simple-talk.com/sql/t-sql-programming/sql-server-cte-basics/*.

FOREIGN KEY constraints relating to a UNIQUE constraint instead of a PRIMARY KEY constraint

Though it is definitely a fringe case, it is allowable for a FOREIGN KEY constraint to reference the columns in a UNIQUE constraint as well as a PRIMARY KEY one. This is often done for a code of some sort, which legitimately would have made a reasonable primary key, and you want to validate the code's value in a table.

As an example, consider a table of colors for products:

Click here to view code image

```
CREATE TABLE Examples.Color
(
      ColorId   int NOT NULL CONSTRAINT PKColor PRIMARY KEY,
      ColorName varchar(30) NOT NULL CONSTRAINT AKColor UNIQUE
);
INSERT INTO Examples.Color(ColorId, ColorName)
VALUES (1,'Orange'),(2,'White');
```

Now, a table is created that needs to use the natural key value:

Click here to view code image

```
CREATE TABLE Examples.Product
(
    ProductId int NOT NULL CONSTRAINT PKProduct PRIMARY KEY,
    ColorName varchar(30) NOT NULL
         CONSTRAINT FKProduct_Ref_ExamplesColor
                REFERENCES Examples.Color (ColorName)
);
```

Demonstrate now that everything works as expected:

Click here to view code image

```
INSERT INTO Examples.Product(ProductId,ColorName)
VALUES (1,'Orange');
```

That INSERT statement worked, but the following fails:

Click here to view code image

```
INSERT INTO Examples.Product(ProductId,ColorName)
VALUES (2,'Crimson');
```

This returns the following error:

Click here to view code image

```
Msg 547, Level 16, State 0, Line 266
The INSERT statement conflicted with the FOREIGN KEY constraint
"FKProduct_Ref_ExamplesColor". The conflict occurred in
database "ExamBook762Ch2",
table "Examples.Color", column 'ColorName'.
```

Note that since you can reference a UNIQUE constraint, which can contain NULL column(s), the concerns noted in the previous section with composite indexes and NULL values goes both for the referenced and referencing key values in this case. No parent row with NULL is able to be referenced, and no child row with NULL ever fails.

Limiting a column to a set of values

The one last scenario we cover in terms of using constraints is to implement a domain of a set of columns to a set of values. As our example, consider a column that has the size of shirt for a conference attendee. The values for this column would likely be something like S, M, L, XL, and XXL. There are two common methods of implementing this domain of values:

Click here to view code image

```
CREATE TABLE Examples.Attendee
(
      ShirtSize  varchar(8) NULL
);
```

The first is using a simple CHECK constraint:

Click here to view code image

```
ALTER TABLE Examples. Attendee
    ADD  CONSTRAINT CHKAttendee_ShirtSizeDomain
        CHECK  (ShirtSize in ('S', 'M','L','XL','XXL'));
```

Now, the value is checked on the INSERT or UPDATE operations, so if the user misspells 'XL' as 'LX:'

Click here to view code image

```
INSERT INTO Examples.Attendee(ShirtSize)
VALUES ('LX');
```

They are denied:

Click here to view code image

```
   Msg 547, Level 16, State 0, Line 346
The INSERT statement conflicted with the CHECK constraint
"CHKAttendee_ShirtSizeDo-
main".
The conflict occurred in database "ExamBook762Ch2", table
"Examples.Attendee",
column 'ShirtSize'.
```

The problem is, how do you know what the legitimate values are? For many CHECK constraint conditions, this is not too big a deal as the goal is to limit really outlandish values. However, for a domain of values, it can be helpful to coordinate the domain of values with another table.

The second solution is to use a table of values. So you can create:

Click here to view code image

```
CREATE TABLE Examples.ShirtSize
(
     ShirtSize varchar(10) NOT NULL CONSTRAINT PKShirtSize
PRIMARY KEY
);
INSERT INTO Examples.ShirtSize(ShirtSize)
VALUES ('S'),('M'),('L'),('XL'),('XXL');
```

Now drop the CHECK constraint and replace with a FOREIGN KEY constraint:

Click here to view code image

```
ALTER TABLE Examples.Attendee
    DROP CONSTRAINT CHKAttendee_ShirtSizeDomain;
ALTER TABLE Examples.Attendee
```

```
    ADD CONSTRAINT FKAttendee_Ref_ExamplesShirtSize
        FOREIGN KEY (ShirtSize) REFERENCES
Examples.ShirtSize(ShirtSize);
```

Though the error message has changed, the result is the same:

Click here to view code image

```
INSERT INTO Examples.Attendee(ShirtSize)
VALUES ('LX');
```

It fails:

Click here to view code image

```
Msg 547, Level 16, State 0, Line 364
The INSERT statement conflicted with the FOREIGN KEY constraint
"FKAttendee_Ref_ExamplesShirtSize". The conflict occurred in
database "ExamBook762Ch2",
table "Examples.ShirtSize", column 'ShirtSize'.
```

Even in systems that use surrogate keys for primary keys, it isn't atypical to use a natural key for a domain table, depending on how the tools used interact with the data. Using a foreign key gives you easy expandability (such as adding descriptive information to the values by adding additional columns to your domain table), as well as the ability to add new values to the domain without any coding changes.

Write Transact-SQL statements to add constraints to tables

So far in the chapter, we have added many constraints to tables. In this section we review the basics of this process briefly, and then cover a few more advanced aspects of creating and managing constraints.

When creating a table, there are two ways to add a constraint: on the same line with a column declaration, denoting that the constraint pertains to that column, or delimited by a comma, meaning it could reference any of the columns in the table. As an example of the many ways you can add constraints in the declaration, consider the following:

Click here to view code image

```
CREATE TABLE Examples.CreateTableExample
(
    --Uniqueness constraint referencing single column
    SingleColumnKey int NOT NULL CONSTRAINT
PKCreateTableExample PRIMARY KEY,

    --Uniqueness constraint in separate line
    TwoColumnKey1 int NOT NULL,
    TwoColumnKey2 int NOT NULL,
    CONSTRAINT AKCreateTableExample UNIQUE (TwoColumnKey1,
```

```
TwoColumnKey2),

    --CHECK constraint declare as column constraint
    PositiveInteger int NOT NULL
         CONSTRAINT CHKCreateTableExample_PostiveInteger CHECK
(PositiveInteger > 0),

    --CHECK constraint that could reference multiple columns
    NegativeInteger int NOT NULL,
    CONSTRAINT CHKCreateTableExample_NegativeInteger CHECK
(NegativeInteger > 0),

    --FOREIGN KEY constraint inline with column
    FKColumn1 int NULL CONSTRAINT FKColumn1_ref_Table
REFERENCES Tbl (TblId),

    --FOREIGN KEY constraint... Could reference more than one
columns
    FKColumn2 int NULL,
    CONSTRAINT FKColumn2_ref_Table FOREIGN KEY (FKColumn2)
REFERENCES Tbl (TblId)
);
```

In addition, every constraint has the ability to be dropped and added after the table has been created. With this table, we can drop and recreate the PRIMARY KEY constraint with:

Click here to view code image

```
ALTER TABLE Examples.CreateTableExample
    DROP PKCreateTableExample;
ALTER TABLE Examples.CreateTableExample
    ADD CONSTRAINT PKCreateTableExample PRIMARY KEY
(SingleColumnKey);
```

You can do this for every one of the constraint types. However, for the ALTER TABLE commands for CHECK and FOREIGN KEY constraints, you have a few additional choices to deal with data that doesn't match the constraint. UNIQUE and PRIMARY KEY constraints behave like indexes when being enabled, so you can't violate the uniqueness characteristics. Disabling uniqueness constraints will remove the index.

Consider the following table and data:

Click here to view code image

```
CREATE TABLE Examples.BadData
(
        PositiveValue int NOT NULL
);
```

```
INSERT INTO Examples.BadData(PositiveValue)
VALUES (-1),(-2),(-3),(-4);
```

You want to add the following constraint:

Click here to view code image

```
ALTER TABLE Examples.BadData
   ADD CONSTRAINT CHKBadData_PostiveValue CHECK(PositiveValue >
0);
```

But you are greeted with the following message:

Click here to view code image

```
Msg 547, Level 16, State 0, Line 414
The ALTER TABLE statement conflicted with the CHECK constraint
CHKBadData_PostiveValue". The conflict occurred in database
ExamBook762Ch2", table "Examples.BadData", column
'PositiveValue'.
```

From here, you have two choices. You can (ideally) fix the data, or you can create the constraint and leave the bad data. This can be done by specifying WITH NOCHECK which bypasses the data check:

Click here to view code image

```
ALTER TABLE Examples.BadData WITH NOCHECK
   ADD CONSTRAINT CHKBadData_PostiveValue CHECK(PositiveValue >
0);
```

The problem with this approach is twofold. First, you have bad data in the table. So if you run the following statement that sets the value to an existing value, you get something that seems silly as a statement, but is technically done in user code all of the time:

Click here to view code image

```
UPDATE Examples.BadData
SET    PositiveValue = PositiveValue;
```

The data isn't granted immunity, even though it already exists in the table:

Click here to view code image

```
Msg 547, Level 16, State 0, Line 420
The UPDATE statement conflicted with the CHECK constraint
"CHKBadData_PostiveValue".
The conflict occurred in database "ExamBook762Ch2", table
"Examples.BadData",
column 'PositiveValue'.
```

It seems that if you just delete the data that would violate the constraint, and everything

would be great:

Click here to view code image

```
DELETE FROM Examples.BadData
WHERE  PositiveValue <= 0;
```

And from a certain perspective, it is. If you try to insert a non-positive value, it fails. However, even though the constraint now does everything that you expect it to, because the data wasn't checked when you created the constraint, it is considered not *trusted*, which means that SQL Server has never checked to see that the data is correct. You can see if a CHECK constraint is trusted using the following query (note that CHECK constraints are owned by a schema, just like a table, even though you rarely reference them as such):

Click here to view code image

```
SELECT is_not_trusted, is_disabled
FROM   sys.check_constraints --for a FOREIGN KEY, use
sys.foreign_keys
WHERE  OBJECT_SCHEMA_NAME(object_id) = 'Examples'
  and  OBJECT_NAME(object_id) = 'CHKBadData_PostiveValue';
```

Which returns:

```
is_not_trusted is_disabled
-------------- -----------
1              0
```

This shows you how the constraint is not trusted, but it is enabled. Now that you know the data in the table is correct, you can tell the constraint to check the data in the table using the following command:

Click here to view code image

```
ALTER TABLE Examples.BadData WITH CHECK CHECK
 CONSTRAINT CHKBadData_PostiveValue;
```

If you check the constraint now to see if it is trusted, it is. If you want to disable (turn off) a CHECK or FOREIGN KEY constraint, you can use NOCHECK in the ALTER TABLE command:

Click here to view code image

```
ALTER TABLE Examples.BadData
       NOCHECK CONSTRAINT CHKBadData_PostiveValue;
```

After running this, you can see that the constraint has been disabled.

Having a trusted CHECK constraint can be useful for performance. The Query Optimizer can use a trusted constraint in optimizing queries. If a value that is searched for would be illegal for the CHECK constraint predicate, it does not need to even check the physical

data (assuming there is enough data in the table to make it worth optimizing more than just a simple plan). For an example with a reasonable amount of data, we a table in the WideWorldImporters database. Consider that the domain of the OrderId column of the Sales.Invoices table should be 0 to 1,000,000. We might add a constraint such as the following:

Click here to view code image

```
ALTER TABLE Sales.Invoices
    ADD CONSTRAINT CHKInvoices_OrderIdBetween0and1000000
        CHECK (OrderId BETWEEN 0 AND 1000000);
```

Now, consider the following two queries are performed:

```
SELECT *
FROM   Sales.Invoices
WHERE  OrderID = -100;

SELECT *
FROM   Sales.Invoices
WHERE  OrderID = 100;
```

Notice that, despite how similar the queries appear, their query plans are quite different; as seen in Figure 2-1. The first query sees that the value is outside of the legal domain and returns immediately, while the other needs to look at the data.

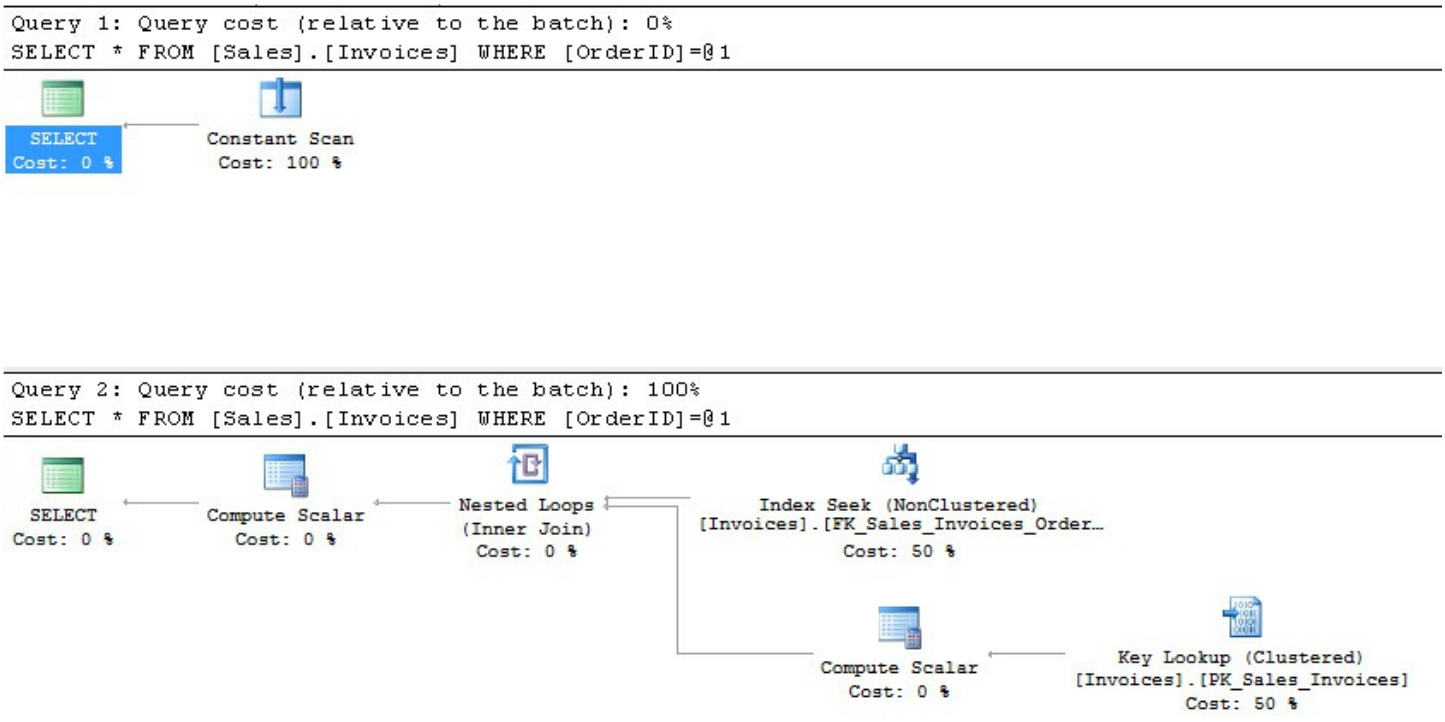


FIGURE 2-1 Comparison of plans, where the first value is eliminated by the CHECK predicate, and the second is not

Need More Review? Altering a Table's Constraints

FOREIGN KEY constraints also have trusted and disabled status with you can find in sys.foreign_keys, and the same syntax that sets CHECK constraints trusted works the same way. There are many settings and uses of the ALTER TABLE command. For more information for review, check the MSDN article here: *https://msdn.microsoft.com/en-us/library/ms190273.aspx*.

Identify results of Data Manipulation Language (DML) statements given existing tables and constraints

An important task for almost any database programmer is to be able to predict the outcome given some Transact-SQL Data Definition Language (DDL) code that sets up a scenario, followed by some DML (Data Manipulation Language) that you need to determine the outcome of. Every time we set up some new concept, the next thing to do is to show it working (one of the great things about working with a declarative interactive language like Transact-SQL).

In this example, we highlight the process of working through a given scenario. Fortunately, when you are taking the exam, questions are all multiple choice, and there is always one answer that is correct for a question of this variety. Imagine you have the following table structure.

Click here to view code image

```
CREATE TABLE Examples.ScenarioTestType
(
    ScenarioTestType varchar(10) NOT NULL CONSTRAINT
PKScenarioTestType PRIMARY KEY
);
CREATE TABLE Examples.ScenarioTest
(
    ScenarioTestId int NOT NULL PRIMARY KEY,
    ScenarioTestType varchar(10) NULL CONSTRAINT
CHKScenarioTest_ScenarioTestType
                                        CHECK ScenarioTestType
IN ('Type1','Type2'))

);
ALTER TABLE Examples.ScenarioTest
   ADD CONSTRAINT FKScenarioTest_Ref_ExamplesScenarioTestType
       FOREIGN KEY (ScenarioTestType) REFERENCES
Examples.ScenarioTestType;
```

Now, after contemplating what is going on in the DDL, you need to work through something like the following DML. As an exercise, consider how many rows are inserted into Examples.ScenarioTest when these statements are performed in a single batch:

Click here to view code image

```
INSERT INTO Examples.ScenarioTest(ScenarioTestId,
ScenarioTestType)
VALUES (1,'Type1');
INSERT INTO Examples.ScenarioTestType(ScenarioTestType)
VALUES ('Type1');
INSERT INTO Examples.ScenarioTest(ScenarioTestId,
ScenarioTestType)
VALUES (1,'Type1');
INSERT INTO Examples.ScenarioTest(ScenarioTestId,
ScenarioTestType)
VALUES (1,'Type2');
INSERT INTO Examples.ScenarioTest(ScenarioTestId,
ScenarioTestType)
VALUES (2,'Type1');
INSERT INTO Examples.ScenarioTests(ScenarioTestId,
ScenarioTestType)
VALUES (3,'Type1');
```

Anything is possible, and while questions aren't created to trick you, they are tricky if you do not have a grasp on some of the finer details of the topics covered, and are set up to make you think and pay attention to details. In this batch, first you need to know that a CHECK or FOREIGN KEY constraint error does not stop the batch, so the answer is not just simply zero.

Next, you need to consider each statement in order. The first statement violates the FOREIGN KEY constraint, but not the CHECK constraint.

Click here to view code image

```
INSERT INTO Examples.ScenarioTest(ScenarioTestId,
ScenarioTestType)
VALUES (1,'Type1');
```

The next line adds a row to the domain table, so doesn't change the number of rows in the ScenarioTest table.

Click here to view code image

```
INSERT INTO Examples.ScenarioTestType(ScenarioTestType)
VALUES ('Type 1')
```

The next INSERT statement now succeeds, as it passes the test of the FOREIGN KEY and the CHECK constraints, and the primary key value is not duplicate (since it is the first row). So you have 1 row in the table.

Click here to view code image

```
INSERT INTO Examples.ScenarioTest(ScenarioTestId,
```

```
ScenarioTestType)
VALUES (1,'Type1');
```

The next INSERT statement violates the domain table's FOREIGN KEY constraint.

Click here to view code image

```
INSERT INTO Examples.ScenarioTest(ScenarioTestId,
ScenarioTestType)
VALUES (1,'Type2');
```

The next insert is the same ScenarioTestType that worked prior, and the new ScenarioTestId does not violate the PRIMARY KEY constraint. So there are 2 rows in the table.

Click here to view code image

```
INSERT INTO Examples.ScenarioTest(ScenarioTestId,
ScenarioTestType)
VALUES (2,'Type1');
```

The last INSERT statement looks like it works, but the table name is ScenarioTests (plural) in the INSERT statement, so it fails (or at least does not insert into the table that the question is about if there is a table by that name in the database):

Click here to view code image

```
INSERT INTO Examples.ScenarioTests(ScenarioTestId,
ScenarioTestType)
VALUES (3,'Type1');
```

So, there are 2 rows that are returned from the DML. Admittedly, it is much easier when you can use Management Studio to test your expectations than under the stress of the exam timer.

Identify proper usage of PRIMARY KEY constraints

Choosing a primary key during the design phase of a project is generally pretty straightforward. Many designers use the simplest candidate key chosen during design. For example, consider you have a table that defines companies that you do business with:

Click here to view code image

```
CREATE SCHEMA Examples;
GO
CREATE TABLE Examples.Company
(
    CompanyName   nvarchar(50) NOT NULL CONSTRAINT PKCompany
PRIMARY KEY,
    CompanyURL nvarchar(max) NOT NULL
);
```

Insert a few rows into the table to show that the table works. This example is very straightforward, but as this section progresses and we start to alter the table to do interesting things, it becomes more and more interesting to test a few rows.

Click here to view code image

```
INSERT INTO Examples.Company(CompanyName, CompanyURL)
VALUES ('Blue Yonder
Airlines','http://www.blueyonderairlines.com/'),
       ('Tailspin Toys','http://www.tailspintoys.com/');
```

Now, check out the data that has been created.

```
SELECT *
FROM   Examples.Company;
```

You see the rows you expected from the INSERT statement.

Click here to view code image

```
CompanyName              CompanyURL
------------------------ ---------------------------------------
Blue Yonder Airlines     http://www.blueyonderairlines.com/
Tailspin Toys            http://www.tailspintoys.com/
```

Note that this, by default, makes the CompanyName the clustered index of the table. But is a value that has an upper limit on data size of 100 bytes (50 characters * 2 bytes per character for typical Unicode characters) really the best choice? This limits the number of index keys on a page for all indexes you add as well. And if this key is used as a foreign key, if this value needed changing, it would require changes to any child tables as well. The choice of how to use the PRIMARY KEY constraint is as much about performance as it is about the integrity of the data.

Another choice you might consider is a more derived key value, such as: CompanyCode that is either a strictly formatted value (such as char(3) and a domain of numbers or letters, like for a company named “Fourth Coffee” it might be “4CF”, much like the stock exchange ticker code is for a company), or even a 10-20 character string value that is kept very short, like “4Coffee”, or something similar). Of course while this solves the length issue, if the name of the company changes to “Tailspin Toys” you still need to change the value or end up with a confusing CompanyCode value.

In typical practice, most people tend to use something like an ever increasing integer type for their primary key, using this as a stand-in for the natural key (commonly referred to as a *surrogate key*.) This is generally great because what makes a great surrogate key overlaps nicely with what makes a great clustering key (monotonically increasing, never needs changing, small data size), works great for foreign keys references, and always ends up with a single key column.

So where CompanyName was the key of the PRIMARY KEY constraint before, we now

make an integer-valued column named CompanyId. Using an integer surrogate key, it is typical to use either a column with the IDENTITY property, or a DEFAULT constraint that uses the result of a SEQUENCE object. For the Examples.Company table, first, use an IDENTITY column which you specify in the following manner:

Click here to view code image

```
DROP TABLE IF EXISTS Examples.Company;
CREATE TABLE Examples.Company
(
    CompanyId     int NOT NULL IDENTITY(1,1) CONSTRAINT
PKCompany PRIMARY KEY,
    CompanyName   nvarchar(50) NOT NULL CONSTRAINT AKCompany
UNIQUE,
    CompanyURL nvarchar(max) NOT NULL
);
```

The IDENTITY setting has two parameters, the first being the *seed* value, or the starting point when the values start generating. The second is the *step* value, which indicates the difference between the previous value and the next value. Now, let's create some sample data:

Click here to view code image

```
INSERT INTO Examples.Company(CompanyName, CompanyURL)
VALUES ('Blue Yonder
Airlines','http://www.blueyonderairlines.com/'),
       ('Tailspin Toys','http://www.tailspintoys.com/');
```

Before we look at the data, let's take a look at what happens if a row fails to be created. So we violate the alternate key UNIQUE constraint.

Click here to view code image

```
INSERT INTO Examples.Company(CompanyName, CompanyURL)
VALUES ('Blue Yonder
Airlines','http://www.blueyonderairlines.com/');
```

This gives the following error:

Click here to view code image

```
Msg 2627, Level 14, State 1, Line 53
Violation of UNIQUE KEY constraint 'AKCompany'. Cannot insert
duplicate key in object
'Examples.Company'. The duplicate key value is (Blue Yonder
Airlines).
```

Now insert another row that is not in violation:

Click here to view code image

```
INSERT INTO Examples.Company(CompanyName, CompanyURL)
VALUES ('Northwind
Traders','http://www.northwindtraders.com/');
```

Looking at the rows in the table, you see that there is a value missing from the expected sequence of values. The IDENTITY value is generated before the value fails, and for concurrency purposes, the value is not returned to be reused.

If you need to get the value of the IDENTITY value after the insert, you can use SCOPE_IDENTITY() to get the value in the current scope, or @@IDENTITY to get the last value for the connection. SCOPE_IDENTITY() is generally preferred.

```
SELECT *
FROM   Examples.Company;
```

Which returns:

Click here to view code image

```
CompanyId   CompanyName           CompanyURL
----------- --------------------- ----------------------------
---------
1           Blue Yonder
Airlines  http://www.blueyonderairlines.com/
2           Tailspin Toys         http://www.tailspintoys.com/
4           Northwind
Traders     http://www.northwindtraders.com/
```

You cannot specify values for an IDENTITY column without using SET IDENTITY INSERT <tableName> ON first, then doing your INSERT and turning it off. So, you can go back and add CompanyId = 3, but the general goal of using a surrogate key is to not care about the value of the keys. You cannot modify the value in a column that has the IDENTITY property.

Note that you can add a column with the IDENTITY property, but you cannot control the data being inserted into the new column. If the value matters to you (typically because you are replacing another surrogate value), the typical method is to duplicate the table by renaming the table (using the sp_rename system stored procedure) and making a copy of the table and loading existing data.

A second method of generating a numeric surrogate key is to use a SEQUENCE object. Using the same base table structure, we use a DEFAULT constraint to get the next value for the SEQUENCE object.

Click here to view code image

```
DROP TABLE IF EXISTS Examples.Company;
DROP SEQUENCE IF EXISTS Examples.Company_SEQUENCE;

CREATE SEQUENCE Examples.Company_SEQUENCE AS INT START WITH 1;
```

```
CREATE TABLE Examples.Company
(
    CompanyId     int NOT NULL CONSTRAINT PKCompany PRIMARY KEY
                               CONSTRAINT DFLTCompany_CompanyId
DEFAULT
                                          (NEXT VALUE FOR
Examples.Company_SEQUENCE),
    CompanyName   nvarchar(50) NOT NULL CONSTRAINT AKCompany
UNIQUE,
    CompanyURL nvarchar(max) NOT NULL
);
```

The same INSERT statement from before works great:

Click here to view code image

```
INSERT INTO Examples.Company(CompanyName, CompanyURL)
VALUES ('Blue Yonder
Airlines','http://www.blueyonderairlines.com/'),
       ('Tailspin Toys','http://www.tailspintoys.com/');
```

But now you can insert your own value into the CompanyId column, ideally after fetching the value from the SEQUENCE object.

Click here to view code image

```
DECLARE @CompanyId INT = NEXT VALUE FOR
Examples.Company_SEQUENCE;


INSERT INTO Examples.Company(CompanyId, CompanyName,
CompanyURL)
VALUES (@CompanyId, 'Northwind
Traders','http://www.northwindtraders.com/');
```

Using a SEQUENCE object to create surrogate key values has advantages over IDENTITY columns, but it does require some discipline around values. While IDENTITY values don't guarantee uniqueness, once you create the IDENTITY column, you only end up with duplicates by inserting the duplicate values or using DBCC CHECKIDENT to reseed the value. This method using a SEQUENCE object requires that you fetch the value using NEXT VALUE FOR, or use sp_sequence_get_range to get a range of values that you can work with.

A final alternative method of creating a surrogate primary key value is using a Globally Unique Identifier (GUID) using a uniqueidentier column. GUIDs are great surrogate key values in terms of programmability, but do have a couple of issues when used as a clustering key. First GUID values are fairly large, using 16 bytes of storage (and they are 36 characters when needing to type). They are not monotonically increasing, so they can tend to increase page splits (there is a NEWSEQUENTIALID() function you can use to

generate sequential GUIDs, but they are not necessarily sequential with existing data after a server reboot). The code looks very much like the SEQUENCE example with a DEFAULT constraint:

Click here to view code image

```
DROP TABLE IF EXISTS Examples.Company;
CREATE TABLE Examples.Company
(
    CcompanyId     uniqueidentifier NOT NULL CONSTRAINT
PKCompany PRIMARY KEY
                                  CONSTRAINT DFLTCompany_CompanyId
DEFAULT (NEWID()),
    CompanyName   nvarchar(50) NOT NULL CONSTRAINT AKCompany
UNIQUE,
    CompanyURL nvarchar(max) NOT NULL
);
```

Now you can use the DEFAULT constraint, create your own GUID, or let the client generate the GUID value.

The primary reason for using a surrogate key for your primary key is to make implementation simpler for tooling, and in some cases to make the database easier to follow. For example, say you had a table of driver's licenses:

Click here to view code image

```
CREATE TABLE Examples.DriversLicense
(
        Locality char(10) NOT NULL,
        LicenseNumber varchar(40) NOT NULL,
        CONSTRAINT PKDriversLicense PRIMARY KEY (Locality,
LicenseNumber)

);
```

If you use this as a FOREIGN KEY reference, like in a table of employee driver licenses table:

Click here to view code image

```
CREATE TABLE Examples.EmployeeDriverLicense
(
        EmployeeNumber char(10) NOT NULL, --Ref to Employee
table
        Locality char(10) NOT NULL, --Ref to DriversLicense
table
        LicenseNumber varchar(40) NOT NULL, --Ref to
DriversLicense table
        CONSTRAINT PKEmployeeDriversLicense PRIMARY KEY
```

```
                    (EmployeeNumber, Locality, LicenseNumber)
);
```

Now, say you need to use this key in the key of another table, which has three key columns also. Not only is this messy, but it begins to get rather confusing. While some manner of role-naming helps (changing Locality to DriversLicenceLocality, for example), using a surrogate key changes the table to something easier to follow:

Click here to view code image

```
CREATE TABLE Examples.DriversLicense
(
        DriversLicenseId int CONSTRAINT PKDriversLicense
PRIMARY KEY,
        Locality char(10) NOT NULL,
        LicenseNumber varchar(40) NOT NULL,
        CONSTRAINT AKDriversLicense UNIQUE (Locality,
LicenseNumber)
);
CREATE TABLE Examples.EmployeeDriverLicense
(
        EmployeeDriverLicenseId int NOT NULL
                CONSTRAINT PKEmployeeDriverLicense PRIMARY
KEY,
        EmployeeId int NOT NULL, --Ref to Employee table
        DriversLicenseId int NOT NULL, --Ref to DriversLicense
table
        CONSTRAINT AKEmployeeDriverLicense UNIQUE (EmployeeId,
DriversLicenseId)
);
```

It's easier to read, and easier to understand where the key columns come from at a glance. It is important to realize that the DriversLicenseId actually represents the Locality and LicenseNumber in the EmployeeDriverLicense table. The meaning of the table doesn't change, it is just implemented differently.

> **Need More Review? Primary Key Constraints**
>
> See the following article on MSDN for more details about creating PRIMARY KEY constraints *https://msdn.microsoft.com/en-us/library/ms189039.aspx*.

Skill 2.2 Create stored procedures

Stored procedures are coded objects that are used to bundle together Transact-SQL calls into one simple call from a client. There are, in SQL Server 2016, three versions of stored procedures. All of them are compiled and stored on the server, but each have different coding and execution limitations:

- **Interpreted SQL** These objects are compiled into a format that are then interpreted, one line at a time, by the execution engine. This is the classic version of a stored procedure that has been around since version 1.0.
- **CLR (Common Language Runtime)** These are procedures that are written in a .NET language and have a Transact-SQL calling mechanism.
- **Natively Compiled SQL** These stored procedures are written in Transact-SQL, but are compiled into a C language module. They can only access memory optimized TABLE objects and other natively compiled objects. Natively compiled objects are similar to interpreted SQL stored procedures in appearance, but have many limitations.

While each of the types of STORED PROCEDURE (and all coded object type) are different, they all look to Transact-SQL the same. You can run the code, or use the objects in queries in the same manner with few limitations.

This chapter is focused on interpreted SQL stored procedures, but it is important to understand what the CLR objects are. Using managed, .NET code you can create STORED PROCEDURE objects that behave and are called exactly like ones created in interpreted SQL. The .NET code used has access to everything that the interpreted code does as well. These objects are not created using Management Studio in Transact-SQL, but are built in Visual Studio like other .NET programs.

Natively compiled objects are reviewed in Skill 3.4, but many of the topics in this section apply.

We also limit our focus to user-managed stored procedures in a user database. There are system stored procedures in the master database, and in the resource database, which is a special, hidden read-only database with additional system objects.

This section covers how to:

- Design stored procedure components and structure based on business requirements
- Implement input and output parameters
- Implement table-valued parameters
- Implement return codes
- Streamline existing stored procedure logic
- Implement error handling and transaction control logic within stored procedures

Design stored procedure components and structure based on business requirements

As this section moves forward, we cover several of the most important details about using stored procedures, parameters, and error handling. However, in this section, the focus is on why to use stored procedures, and a few examples that include one or more of the concepts that are covered in the rest of this section in more detail.

There are two common philosophies regarding the use of STORED PROCEDURE objects:

- Using stored procedures as a complete encapsulation layer between the user and the database. This is a prevalent way of thinking for many architects (and a smoother path to using the new natively compiled stored procedures).
- Using stored procedures only to encapsulate complex calls. This utilization is typical for people who use tools to generate their access layer, and only use stored procedures when it is impossible to do from the user interface. Other uses are in building a report interface when simple ad-hoc queries do not suffice.

The biggest difference between the philosophies is that in the complete encapsulation version, you end up with a mix of very simple and very complex stored procedures. Hence we look at some simple procedures as well as an example of a complex one.

The form of a stored procedure is generally quite straightforward. The basic structure is:

Click here to view code image

```
CREATE PROCEDURE SchemaName.ObjectName

[WITH options]
[FOR REPLICATION]
    @Parameter1 datatype,
    @Parameter2 datatype = 'Optional Default',
    @Parameter3 datatype = NULL
AS
    1 or more Transact-SQL statements;
```

STORED PROCEDURE objects are schema-bound, just like TABLE objects, and all of the constraint types, Hence they follow the same rules for naming objects and name space, with the addition that STORED PROCEDURE object names should not be prefixed with SP, as this denotes a system-stored procedure that SQL Server checks for in the master database first.

Parameters are for sending data to, and receiving data from, a stored procedure. Any data type can be used for a parameter, and only very few have limitations (table-valued parameters must be read-only, and cursors can only be output parameters.) Parameters behave for the most part just like variables do in the body of the procedure, with values provided either by the caller or the default, including being changeable by the code of the stored procedure. If no default is provided, the parameter must be specified.

There are very few limitations on what Transact-SQL can be used in a stored procedure.

One example are statements that need to be the first statement in a batch, like CREATE SCHEMA, CREATE PROCEDURE, CREATE VIEW, etc. (and the ALTER versions of the same). You cannot use the USE command to change database context, and there are a few SET commands pertaining to query execution. However, using dynamic SQL (putting your query into a string variable and using EXECUTE or sp_executesql to run the code) allows you to run any code at all.

The options you can specify are:

- **WITH ENCRYPTION** Encrypts the entry in sys.syscomments that contains the text of the STORED PROCEDURE create statement.
- **WITH RECOMPILE** Specifies that a plan is not cached for the procedure, so it is recompiled for every execution.
- **WITH EXECUTE AS** Let's you change the security context that the procedure is executed under.
- **FOR REPLICATION** Indicates that this is a procedure that is specifically created for replication.

> **Note The CREATE PROCEDURE statement**
>
> For more details about the CREATE PROCEDURE statement, check here on the MSDN site: *https://msdn.microsoft.com/en-us/library/ms187926.aspx*. While we will not user the WITH EXECUTE AS clause in the stored procedure examples, it will be used when triggers are created in Skill 2.4.

In the first example, consider the business requirement that the user needs to be able to create, and remove rows from a simple table. If the table is defined as:

Click here to view code image

```
CREATE TABLE Examples.SimpleTable
(
    SimpleTableId int NOT NULL IDENTITY(1,1)
                  CONSTRAINT PKSimpleTable PRIMARY KEY,
    Value1   varchar(20) NOT NULL,
    Value2   varchar(20) NOT NULL
);
```

Then we could create the three stored procedures shown in Listing 2-1:

LISTING 2-1 Three simple stored procedures to insert, update, and delete data

Click here to view code image

```
CREATE PROCEDURE Examples.SimpleTable_Insert
```

```
    @SimpleTableId int,
    @Value1  varchar(20),
    @Value2  varchar(20)
AS
    INSERT INTO Examples.SimpleTable(Value1, Value2)
    VALUES (@Value1, @Value2);
GO

CREATE PROCEDURE Examples.SimpleTable_Update
    @SimpleTableId int,
    @Value1  varchar(20),
    @Value2  varchar(20)
AS
    UPDATE Examples.SimpleTable
    SET Value1 = @Value1,
        Value2 = @Value2
    WHERE SimpleTableId = @SimpleTableId;
GO
CREATE PROCEDURE Examples.SimpleTable_Delete
    @SimpleTableId int,
    @Value  varchar(20)
AS
    DELETE Examples.SimpleTable
    WHERE SimpleTableId = @SimpleTableId
GO
```

As you can see, the body of code for each STORED PROCEDURE is just one simple Transact-SQL query. The power of STORED PROCEDURE objects we are applying here is providing a strict user interface that you have complete control over.

There are three ways you can return data to a user from a stored procedure. Output parameters and return codes are covered later, but the first and most common is by using one or more result sets (Generally one result set is desirable. You can get the metadata of the first result set using the system stored procedure: sp_describe_first_result_set).

For example, you could create a stored procedure to return all of the data in the Examples.SimpleTable, ordered by Value1:

Click here to view code image

```
CREATE PROCEDURE Examples.SimpleTable_Select
AS
   SELECT SimpleTableId, Value1, Value2
   FROM Examples.SimpleTable
   ORDER BY Value1;
```

You can also return multiple result sets to the client, though again, it is generally desirable to return a single result set:

Click here to view code image

```
CREATE PROCEDURE Examples.SimpleTable_SelectValue1StartWithQorZ
AS
   SELECT SimpleTableId, Value1, Value2
   FROM Examples.SimpleTable
   WHERE Value1 LIKE 'Q%'
   ORDER BY Value1;

   SELECT SimpleTableId, Value1, Value2
   FROM Examples.SimpleTable
   WHERE Value1 LIKE 'Z%'
   ORDER BY Value1 DESC;
```

Another ability is to return a variable number of result sets. For example, say the requirement is to allow a user to search the table for rows where Value1 starts with 'Q' or 'Z' and the query is performed on a weekday. Whereas we could have done the INSERT or UPDATE statements from Listing 2-1, without the stored procedure you could not fulfill the business requirement in Transact-SQL code. For this (obviously contrived) requirement, you could write the following stored procedure which returns either 1 or 0 result sets:

Click here to view code image

```
CREATE PROCEDURE Examples.SimpleTable_SelectValue1StartWithQorZ
AS
  IF DATENAME(weekday,getdate()) NOT IN ('Saturday','Sunday')
     SELECT SimpleTableId, Value1, Value2
     FROM Examples.SimpleTable
     WHERE  Value1 LIKE '[QZ]%';
```

There are more elegant ways of accomplishing this requirement, particularly by throwing an error to alert the user why nothing is returned, but we cover this later in this section. In the following sections covering the specific skills of writing a STORED PROCEDURE, the examples get more and more complex.

One interesting characteristic of STORED PROCEDURE code is that it can reference objects that do not exist. For example, you could code:

Click here to view code image

```
CREATE PROCEDURE Examples.ProcedureName
AS
SELECT ColumnName From Bogus.TableName;
```

This compiles, regardless of the existence of Bogus.TableName. The compilation process of creating the procedure stores the reference by name and by internal id values when available. When it doesn't have the internal id values, it tries to fetch them at execution time. If the object does not exist at runtime, you get an error telling you "Invalid object name 'Bogus.TableName'."

One last fundamental about developing STORED PROCEDURE objects to understand. Configurations you set on the connection using the SET command in the body of the STORED PROCEDURE only pertains to the statements in the STORED PROCEDURE. The most common example is the SET NOCOUNT ON setting that suppresses the "Rows Affected" messages, but others such as ANSI WARNINGS can definitely be useful. So in the following procedure:

Click here to view code image

```
CREATE PROCEDURE Examples.SimpleTable_Select
AS
   SET NOCOUNT ON;
   SELECT SimpleTableId, Value1, Value2
   FROM Examples.SimpleTable
   ORDER BY Value1;
```

If you perform this procedure, no "Rows Affected" would be returned, but if you did an INSERT statement in the same connection it would show the messages, unless you turned them off in the connection. However, silencing the messages from the calling connection silences them in STORED PROCEDURE calls. Because of this, you need to make sure and set any SET options that you rely on in the body of the object.

In this initial section, the main aspect of stored procedure writing we wanted to review was that stored procedures are just compiled batches of Transact-SQL code that you can use to do almost anything, allow you as the database developer to build a set of stored procedures to do almost any task.

Implement input and output parameters

Parameters allow you to pass data in and out of a STORED PROCEDURE object. To the code, they are very similar to variables, and can have their values changed as needed. Whether or not you see the changes to the value of the parameters after running the procedure is based on how you declare, and then call, the STORED PROCEDURE.

The following examples use this table as reference.

Click here to view code image

```
CREATE TABLE Examples.Parameter
(
    ParameterId int NOT NULL IDENTITY(1,1) CONSTRAINT
PKParameter PRIMARY KEY,
    Value1   varchar(20) NOT NULL,
    Value2  varchar(20) NOT NULL,
)
```

In the first iteration, it is simple input parameters for the two value columns:

Click here to view code image

```
CREATE PROCEDURE Examples.Parameter_Insert
    @Value1 varchar(20) = 'No entry given',
    @Value2 varchar(20) = 'No entry given'
AS
    SET NOCOUNT ON;
    INSERT INTO Examples.Parameter(Value1,Value2)
    VALUES (@Value1, @Value2);
```

Now, in the following code block, we show the various ways you can run this code in Transact-SQL:

Click here to view code image

```
--using all defaults
EXECUTE Examples.Parameter_Insert;

--by position, @Value1 parameter only
EXECUTE Examples.Parameter_Insert 'Some Entry';

--both columns by position
EXECUTE Examples.Parameter_Insert 'More Entry','More Entry';

-- using the name of the parameter (could also include
@Value2);
EXECUTE Examples.Parameter_Insert @Value1 = 'Other Entry';

--starting positionally, but finishing by name
EXECUTE Examples.Parameter_Insert 'Mixed Entry', @Value2 =
'Mixed Entry';
```

Once you start filling in parameters by name, you must continue to or you will receive an error. For example, if you attempt:

Click here to view code image

```
EXECUTE Examples.Parameter_Insert @Value1 = 'Remixed Entry',
'Remixed Entry';
```

It causes the following error:

Click here to view code image

```
Msg 119, Level 15, State 1, Line 736
Must pass parameter number 2 and subsequent parameters as
'@name = value'. After the
form '@name = value' has been used, all subsequent parameters
must be passed in the form
'@name = value'.
```

If you want to get the value of a parameter after it has been modified in the STORED

PROCEDURE, you define the parameter as OUTPUT. In the alteration of the Examples.Parameter_Insert stored procedure, use the UPPER and LOWER functions to modify the two @Value parameters, and then retrieve the value of SCOPE_IDENTITY() system function to get the key value of the ParameterId column that has been created:

Click here to view code image

```
ALTER PROCEDURE Examples.Parameter_Insert
    @Value1 varchar(20) = 'No entry given',
    @Value2 varchar(20) = 'No entry given' OUTPUT,
    @NewParameterId int = NULL OUTPUT
AS
    SET NOCOUNT ON;
    SET @Value1 = UPPER(@Value1);
    SET @Value2 = LOWER(@Value2);

    INSERT INTO Examples.Parameter(Value1,Value2)
    VALUES (@Value1, @Value2);

    SET @NewParameterId = SCOPE_IDENTITY();
```

Next we call the procedure using the variables as the parameter value. Set the @NewParameterId value to a value that could not be returned. Note that the parameter is configured as OUTPUT on the EXECUTE statement as well (and if you try to declare a non-OUTPUT parameter as OUTPUT, it gives you an error telling you that the parameter is not an OUTPUT parameter).

Click here to view code image

```
DECLARE @Value1 varchar(20) = 'Test',
        @Value2 varchar(20) = 'Test',
        @NewParameterId int = -200;

EXECUTE Examples.Parameter_Insert @Value1 = @Value1,
                                  @Value2 = @Value2 OUTPUT,
                                  @NewParameterId =
@NewParameterId OUTPUT;

SELECT @Value1 as Value1, @Value2 as Value2, @NewParameterId as
NewParameterId;

SELECT *
FROM Examples.Parameter
WHERE ParameterId = @newParameterId;
```

This returns the following, showing that the @Value2 parameters values did change, as the row inserted has an all uppercase, and all lowercase versions of the parameter values. However, the variable values did not change, unlike the @Value2 and @NewParameterId

values.

Click here to view code image

```
Value1               Value2               NewParameterId
-------------------- -------------------- --------------
Test                 test                 7

ParameterId Value1               Value2
----------- -------------------- --------------------
7           TEST                 test
```

Using output parameters is the only way to return data to a variable that is not an integer, and is the only way to return more than one value to a variable, no matter the type.

Implement table-valued parameters

Table-valued parameters allow you need to pass more than a simple scalar value to a procedure. The reasons to do this generally fall into two categories:

- The user wants to pick a set of rows to filter a set that could not easily be done using scalar valued parameters.
- To create a procedure interface that allows you to create more than a single row in the table in a natural way.

For our example, we use a small table that has a few columns, and start with a few rows as well.

Click here to view code image

```
CREATE TABLE Examples.Machine
(
    MachineId   int NOT NULL CONSTRAINT PKMachine PRIMARY KEY,
    MachineNumber char(3) NOT NULL CONSTRAINT AKMachine UNIQUE,
    Description varchar(50) NOT NULL
);
INSERT INTO Examples.Machine(MachineId, MachineNumber,
Description)
VALUES (1,'001','Thing1'),(2,'002','Thing2'),
(3,'003','Thing3');
```

Now, consider the case where the user wants to target a few rows. One method that was used for many years was to pass in a comma-delimited list. There were many ways of splitting the string, but SQL Server 2016 added a new system function STRING_SPLIT(). So we could pass a variable value such as ‘1,3’ to get three specific rows in the table.

Click here to view code image

```
CREATE PROCEDURE Examples.Machine_MultiSelect
    @MachineList varchar(200)
```

```
AS
    SET NOCOUNT ON;
    SELECT Machine.MachineId, Machine.MachineNumber
    FROM   Examples.Machine
            JOIN STRING_SPLIT(@MachineList,',') AS StringList
                ON StringList.value = Machine.MachineId;
```

While this works, and it still a very useful way of passing a pseudo-table valued parameter in cases where the caller is not able to use a table-valued parameter, the more efficient method of passing in a table of values is to pass a table object.

> **Note The SQL Server 2016 database engine**
>
> It is great to be up to date about the latest improvements in SQL Server 2016 database engine, particularly Transact-SQL, so if you see STRING_SPLIT(), you won't be confused as to whether it is a trick question or not. The entire list of improvements in the database engine is located here: *https://msdn.microsoft.com/en-us/library/bb510411.aspx*.

To use a table-valued parameter that is actually a true table object, we start by creating a USER DEFINED TYPE with the table data type. We use a generic structure, because this is a fairly generic usage to fetch a specific set of rows. Note that types are schema-owned, but they are not objects, so their names need not be different from the pool of objects.:

Click here to view code image

```
CREATE TYPE Examples.SurrogateKeyList AS table
(
    SurrogateKeyId int PRIMARY KEY --note you cannot name
constraints for table types

);
```

Now you can use this table type as a parameter to the STORED PROCEDURE. You must define it as READONLY, which means that the rows in the table are static. Hence, unlike other parameter values, if you try to INSERT, UPDATE, or DELETE rows you receive an error. In the following procedure, join the TABLE variable to the permanent TABLE object for the output.

Click here to view code image

```
ALTER PROCEDURE Examples.Machine_MultiSelect
    @MachineList Examples.SurrogateKeyList READONLY
AS
    SET NOCOUNT ON;
    SELECT Machine.MachineId, Machine.MachineNumber
    FROM   Examples.Machine
```

```
            JOIN @MachineList AS MachineList
                ON MachineList.SurrogateKeyId =
Machine.MachineId;
```

Calling this procedure is a bit more complex than using a string, but the code is far more straightforward. For parameter sets that are a little larger, you can even include PRIMARY KEY and UNIQUE constraints when defining the table USER DEFINED TYPE, so using the table in a JOIN will have a better chance of using an index.

Click here to view code image

```
DECLARE @MachineList Examples.SurrogateKeyList;
INSERT INTO @MachineList (SurrogateKeyId)
VALUES (1),(3);

EXECUTE Examples.Machine_MultiSelect @MachineList =
@MachineList;
```

Beyond the ability to return a specific set of rows, passing a table of values can be used to create multiple rows in a single call. It is technically possible to do without a table, as you could either use multiple sets of parameters (@MachineId1, @MachineNumber1, @MachineId2, etc), or a complex parameter such as an XML type, but neither is as straightforward as using a table-valued parameter. As with our previous example, we start by creating a table USER DEFINED TYPE, but this time it is defined in a specific manner. We named this USER DEFINED TYPE the same as the TABLE object to reinforce that they are different name spaces, which could be something confusing in an exam question.

Click here to view code image

```
CREATE TYPE Examples.Machine AS TABLE
(
    MachineId int NOT NULL PRIMARY KEY,
    MachineNumber char(3) NOT NULL UNIQUE,
    Description varchar(50) NOT NULL
);
```

Now, create the STORED PROCEDURE to insert rows by using this table type for the parameter:

Click here to view code image

```
CREATE PROCEDURE Examples.Machine_MultiInsert
    @MachineList Examples.Machine READONLY
AS
    SET NOCOUNT ON;
    INSERT INTO Examples.Machine(MachineId, MachineNumber,
Description)
    SELECT MachineId, MachineNumber, Description
    FROM   @MachineList;
```

Now you can call this STORED PROCEDURE after inserting rows into a table variable that works in a very natural manner:

Click here to view code image

```
DECLARE @NewMachineRows Examples.Machine;
INSERT INTO @NewMachineRows (MachineId, MachineNumber,
Description)
VALUES (4,'004','NewThing4'), (5, '005','NewThing5');

EXECUTE Examples.Machine_MultiInsert @MachineList =
@NewMachineRows;
```

Implement return codes

When you run a STORED PROCEDURE object, there is a value returned to the caller if it requests it, and is called a return code (or sometimes, a return value). By default, the return value is 0, but you can change the value using the RETURN statement, which also ends the running of the procedure.

As an example, consider this very simple STORED PROCEDURE object that makes no mention of any return value:

Click here to view code image

```
CREATE PROCEDURE SimpleReturnValue
AS
    DECLARE @NoOp int;
```

Running this procedure, you can access the return code from the EXECUTE statement.

Click here to view code image

```
DECLARE @ReturnCode int;
EXECUTE @ReturnCode = SimpleReturnValue;
SELECT @ReturnCode as ReturnCode;
```

You see the output as:

```
-----------
0
```

You can use the RETURN statement to stop execution and send the return code to the caller to know what has occurred. Return codes can be used in addition to outputting an error (as seen in our example), or in conjunction with error handling, as you use in the final section of this section.

As an example, consider the following STORED PROCEDURE object. In the design of the procedure, the procedure creator, when needed, defines a set of return codes. A typical set of return codes that is common is to use positive vales to mean a positive outcome with some additional information. A negative value is an error, and 0 simply means the

procedure ran successfully with no additional information, since that is what you get performing a procedure that has not set the value itself.

Click here to view code image

```
CREATE PROCEDURE DoOperation
(
    @Value  int
)
--Procedure returns via return code:
-- 1 - successful execution, with 0 entered
-- 0 - successful execution
-- -1 - invalid, NULL input
AS
    IF @Value = 0
        RETURN 1;
    ELSE IF @Value IS NULL
        RETURN -1;
    ElSE RETURN 0;
```

Performing this procedure always looks like a success, as there is not any code to throw an error, but the caller can check the return code to determine what has occurred. As an example, if the value is NULL:

Click here to view code image

```
DECLARE @ReturnCode int;
EXECUTE @ReturnCode = DoOperation @Value = NULL;
SELECT  @ReturnCode,
        CASE @ReturnCode WHEN 1 THEN 'Success, 0 Entered'
                         WHEN -1 THEN 'Invalid Input'
                         WHEN 0 THEN 'Success'
        END as ReturnMeaning;
```

You receive the following output:

Click here to view code image

```
            ReturnMeaning
----------- ----------------
-1          Invalid Input
```

Using the return code as the primary way to represent an error to the caller is not the typical pattern for implementation, but it is available when needed.

Streamline existing stored procedure logic

One of the primary values of using STORED PROCEDURE objects as your interface to your data is that you can fix poorly-written code without changing compiled code of the interface. The biggest win is that often it is non-trivial code that generates the Transact-

SQL in the procedural programming language.

For example, say you have the following table and seed data:

Click here to view code image

```
CREATE TABLE Examples.Player
(
    PlayerId    int NOT NULL CONSTRAINT PKPlayer PRIMARY KEY,
    TeamId      int NOT NULL, --not implemented reference to
Team Table
    PlayerNumber char(2) NOT NULL,
    CONSTRAINT AKPlayer UNIQUE (TeamId, PlayerNumber)
)
INSERT INTO Examples.Player(PlayerId, TeamId, PlayerNumber)
VALUES (1,1,'18'),(2,1,'45'),(3,1,'40');
```

A programmer has written the following procedure shown in Listing 2-2 to fetch a player with a given number on any team, but did not understand how to write a set-based query.

LISTING 2-2 Overly complex stored procedure to fetch rows based on a simple filter

Click here to view code image

```
CREATE PROCEDURE Examples.Player_GetByPlayerNumber
(
    @PlayerNumber char(2)
) AS
    SET NOCOUNT ON;
    DECLARE @PlayerList TABLE (PlayerId int NOT NULL);

    DECLARE @Cursor cursor,
            @Loop_PlayerId int,
            @Loop_PlayerNumber char(2)

    SET @cursor = CURSOR FAST_FORWARD FOR ( SELECT PlayerId,
PlayerNumber
                                            FROM   Examples.Pla

    OPEN @cursor;
    WHILE (1=1)
      BEGIN
            FETCH NEXT FROM @Cursor INTO @Loop_PlayerId,
@Loop_PlayerNumber
            IF @@FETCH_STATUS <> 0
                BREAK;

            IF @PlayerNumber = @Loop_PlayerNumber
```

```
                INSERT INTO @PlayerList(PlayerId)
                VALUES (@Loop_PlayerId);

      END;

    SELECT Player.PlayerId, Player.TeamId
    FROM   Examples.Player
             JOIN @PlayerList AS PlayerList
                on PlayerList.PlayerId = Player.PlayerId;
```

The EXECUTE statement for this STORED PROCEDURE is exactly what the requirements ask for:

Click here to view code image

```
EXECUTE  Examples.Player_Get @PlayerNumber = '18';
```

As is the output:

```
PlayerId    TeamId
----------- -----------
1           1
```

The only problem is that code is way more complex than it needs to be. Understanding the basics of writing code using proper Transact-SQL constructs, the procedure can be written far more simply:

Click here to view code image

```
ALTER PROCEDURE Examples.Player_GetByPlayerNumber
(
    @PlayerNumber char(2)
) AS
    SET NOCOUNT ON

    SELECT Player.PlayerId, Player.TeamId
    FROM   Examples.Player
    WHERE  PlayerNumber = @PlayerNumber;
```

The EXECUTE statement you use to run this STORED PROCEDURE is the same:

Click here to view code image

```
EXECUTE  Examples.Player_GetByPlayerNumber @PlayerNumber =
'18';
```

As is the output:

```
PlayerId    TeamId
----------- -----------
1           1
```

Both procedures return the same amount of data, and for a development server with a few, to a few hundred, rows perform in imperceptibly similar times. However, once you get a realistic amount of data in the table, the performance for the original is very unsatisfactory.

While this is obviously an example of very egregious performance coding issues, things aren't always so obvious. There are many different ways one could create STORED PROCEDURE objects filled with Transact-SQL that is not efficient that are obvious (lots of branching, returning a variable number of result sets based on parameter values), but there are a few that are not quite so obvious with which you should be concerned:

- Parameter type mismatch with data in referenced tables
- Invalid use of functions on search arguments in queries

In the next two sections we take a look at these two considerations in a bit more detail.

Invalid use of functions on search arguments

It is often useful to use system functions to get the current time, to format some data, etc. However, sometimes doing something in a way that seems natural can be bad for performance. For example, consider a stored procedure designed to fetch a list of the games on the current day (showing only the parts of the script that are pertinent):

Click here to view code image

```
CREATE TABLE Game
...
GameStartTime  datetime2(0)
...
CREATE PROCEDURE Game_GetForDate
(
         @SearchDate date
...
     FROM  Game
     WHERE CAST(GameTime AS DATE) = @SearchDate;
```

This is definitely the easiest way to code this, as the CAST expression strips off the time in the GameTime column values and let you compare to the @SearchDate. However, every row that is output from the FROM clause (in this case all rows in the Game table) has to have the GameTime column value converted and checked for each row. If there is an index on the GameTime column, it is not usable.

To fix this, you can change the code to something that looks more complex, such as:

Click here to view code image

```
FROM GAME
WHERE GameTime >= @SearchDate
    AND   GameTime < DATEADD(Day, 1, @SearchDate);
```

But now, the two scalar expressions of @SearchDate and DATEADD(Day, 1, @SearchDate) can be calculated once and used to probe an index to see if it matches for the GameTime column.

Parameter type mismatch

Matching the parameter (or really any variable) to how you are using it is very important. If the sizes don't match, you can lose data. If the types don't match, queries can have to implicitly convert the data type in a query, eliminating the use of an index. The problem is very similar to the problem in the previous section on using functions on search arguments, but at its worst it happens silently in code that looks otherwise correct.

For example, say you have a STORED PROCEDURE object that includes the following (again showing only the parts of the script that are pertinent):

Click here to view code image

```
CREATE TABLE Order
...
    OrderNumber nvarchar(10)

...

CREATE PROCEDURE Order_Search
@OrderNumber int --because a "number" is an integer, one
surmises
...
WHERE OrderNumber = @OrderNumber;
```

Several problems can frequently occur. First, if any order numbers have data that cannot be implicitly converted to an integer (such as 'Order#20'), when that row is reached as rows are being returned, it gives you an error and stop sending back data. It works this way because the integer data type is higher in precedence than nvarchar. In this case, had the OrderNumber column been an integer, the query processor would convert the value in the parameter variable before performing the query. In cases like the Order_Search STORED PROCEDURE object, no index would be used in the search, even if a suitable one existed.

Second, if the data types are incompatible, you can get an immediate error. For example, a datatime2 parameter and an integer column provides an operand-type clash.

> **Need More Review? Datatype conversion and precedence**
>
> Datatype conversion and precedence are important topics to understand. The following two articles on MSDN cover conversion and precedence, respectively: *https://msdn.microsoft.com/en-us/library/ms191530.aspx* and *https://msdn.microsoft.com/en-us/library/ms190309.aspx*.

Implement error handling and transaction control logic within stored procedures

Now we are going to pull the concept of creating a STORED PROCEDURE object together and look at what needs to go into a production worthy stored procedure. So far, the topics have catered to single statement procedures showing one simple concept, but now we are going to get into dealing with multiple statements that modify data. When we start to bundle together multiple modification statements, it becomes important that we are able to make sure that the first statement performed properly before continuing to the next statement.

What makes this difficult is that different types of errors behave differently when performed in different ways, from typical constraints to errors that are thrown by triggers. When an error is caused by a constraint, the batch continues, but if the transaction is rolled back in a TRIGGER object, the batch stops. By building in the proper error handling layer, all errors are treated the same way, which allows you to make sure that one statement has completed successfully.

There are several topics in the process of error handling and transaction control logic that we review in this section:

- **Throwing an error** It is often useful to be able to throw our own error messages to cause the stored procedure code (or really any code) to stop, telling the caller why.
- **Handling an error** In order to manage the code flow after an error has occurred, you need to be able to capture the error and act accordingly.
- **Transaction control logic in your error handling** Transactions are used to control grouping statements together to ensure that multiple statements complete or fail as an atomic unit.

> **Exam Tip Pay special attention to Error Handling**
>
> Like in real life, error handling can be mixed in with other topics. Having knowledge of how different statements may react to error handling, and how to undo work that has been done can be a part of more than just questions about error handling itself.

Throwing an error

In your stored procedure, it is often necessary to tell the caller that there is an issue. Earlier in the chapter, we had a procedure that used return codes to indicate to the caller that there was an issue with the parameter value. There are two methods of throwing an error in Transact-SQL. First is using the THROW statement. THROW lets you specify an error number (50000 or greater, as 49999 and under are system reserved values); a user defined message in plain, Unicode text; and a state value which can be used to send additional

information to the client.

For example, you can perform:

Click here to view code image

```
THROW 50000, 'This is an error message',1;
```

And you get the following output:

Click here to view code image

```
Msg 50000, Level 16, State 1, Line 1115
This is an error message
```

There is another command, RAISERROR, which seemingly does the same thing, with a few subtle differences. First, when specifying an error message, you can only return error number 50000 using RAISERROR. Second, you can change the error level using RAISERROR. There are a few formatting methods you can use with RAISERROR, along with a syntax form that we won't review using custom system error messages. For more information on various forms of using RAISERROR, MSDN has an article that covers this in detail here: *https://msdn.microsoft.com/en-us/library/ms178592.aspx*.) Using typical RAISERROR usage, you can run the following statement:

Click here to view code image

```
RAISERROR ('This is an error message',16,1);
```

You get the same output as with the THROW statement. The big difference between THROW and RAISERROR is how they affect the batch you are running in. THROW stops the batch, and RAISERROR does not. For example, run the following:

Click here to view code image

```
THROW 50000, 'This is an error message',1;
SELECT 'Batch continued'
```

The output is:

Click here to view code image

```
Msg 50000, Level 16, State 1, Line 1117
This is an error message
```

But then run the following:

Click here to view code image

```
RAISERROR ('This is an error message',16,1);
SELECT 'Batch continued'
```

And the output is:

Click here to view code image

```
Msg 50000, Level 16, State 1, Line 1119
This is an error message

---------------
Batch continued
```

So, going to the simple STORED PROCEDURE we had created earlier, you might change it to include a THROW call instead of using a return code for a negative outcome, but it is important to understand what this means to the control of flow. As an example that could easily be an exam question, consider the following stored procedure:

Click here to view code image

```
CREATE PROCEDURE DoOperation
(
    @Value  int
)
AS
    SET NOCOUNT ON;
    IF @Value = 0
        RETURN 1;
    ELSE IF @Value IS NULL
     BEGIN
        THROW 50000, 'The @value parameter should not be
NULL',1;
        SELECT 'Continued to here';
        RETURN -1;
     END
    ELSE RETURN 0;
```

If the following batch is run, what is the output?

Click here to view code image

```
DECLARE @ReturnCode int
EXECUTE @ReturnCode = DoOperation @Value = NULL;
SELECT  @ReturnCode AS ReturnCode;
```

The question is having choices asking if you see the @ReturnCode output, the output ‘Contintued to here’, or just the error message. It turns out that the output is just the error message:

Click here to view code image

```
Msg 50000, Level 16, State 1, Procedure DoOperation, Line 10
The @value parameter should not be NULL
```

If you swap out the THROW statement for the following RAISERROR statement:

Click here to view code image

```
RAISERROR ('The @value parameter should not be NULL',16,1);
```

The output changes to show all three:

Click here to view code image

```
Msg 50000, Level 16, State 1, Procedure DoOperation, Line 11
The @value parameter should not be NULL

-----------------
Continued to here

-----------
-1
```

> **Need More Review? THROW and RAISERROR**
>
> For a more complete coverage of the differences between THROW and RAISERROR, the following article on MSDN is very helpful *https://msdn.microsoft.com/en-us/library/ee677615.aspx*.

Handling an error

Now that we have established how to throw our own error messages, we now need to look at how to handle an error occurring. What makes this difficult is that most errors do not stop processing (an unhandled error from a TRIGGER object is an example of one that ends a batch, as does executing the statement: SET XACT_ABORT ON before your queries that may cause an error, which we discuss in the next section), so when you have a group of modification statements running in a batch without any error handling, they keep running. For example, consider the following table set up to allow you to easily cause an error:

Click here to view code image

```
CREATE TABLE Examples.ErrorTesting
(
    ErrorTestingId int NOT NULL CONSTRAINT PKErrorTesting
PRIMARY KEY,
    PositiveInteger int NOT NULL
         CONSTRAINT CHKErrorTesting_PositiveInteger CHECK
PositiveInteger > 0)
);
```

Now, perform the following five statements, all as a batch:

Click here to view code image

```
INSERT INTO Examples.ErrorTesting(ErrorTestingId,
PositiveInteger)
```

```
VALUES (1,1); --Succeed
INSERT INTO Examples.ErrorTesting(ErrorTestingId,
PositiveInteger)
VALUES (1,1); --Fail PRIMARY KEY violation
INSERT INTO Examples.ErrorTesting(ErrorTestingId,
PositiveInteger)
VALUES (2,-1); --Fail CHECK constraint violation
INSERT INTO Examples.ErrorTesting(ErrorTestingId,
PositiveInteger)
VALUES (2,2); --Succeed
SELECT *
FROM   Examples.ErrorTesting;
```

This returns several error messages, and the two rows that were successfully inserted:

Click here to view code image

```
Msg 2627, Level 14, State 1, Line 1113
Violation of PRIMARY KEY constraint 'PKErrorTesting'. Cannot
insert duplicate key
in object 'Examples.ErrorTesting'. The duplicate key value is
(1).

Msg 547, Level 16, State 0, Line 1116
The INSERT statement conflicted with the CHECK constraint
"CHKErrorTesting_PositiveInteger". The conflict occurred in
database
"ExamBook762Ch2", table "Examples.ErrorTesting", column
'PositiveInteger'.

ErrorTestingId PositiveInteger
-------------- ---------------
1              1
2              2
```

There are two prevalent methods of dealing with these errors to stop the execution. First, use the @@ERROR system function to check the error level after each statement, exiting if so. Second, use the TRY...CATCH construct. TRY...CATCH is by far the easiest and most powerful and modern method, but there are places where checking the error level is still a valid and useful thing to do.

> **Note Undoing changes**
>
> In the following section after we cover the ways to divert the code on an error, we review how the batch that we performed with errors could have all its changes undone.

Using @@ERROR to deal with errors

The @@ERROR system function (also referred to as a global variable, because it is prefixed with @@, though it is technically a system function), tells you the error level of the previous statement.

So, you have to either use it in a Boolean expression, or capture the value immediately after a statement that you are concerned about. You can check how the value changes when you successfully view the value the function returns.

Using the TABLE object we started with in the previous section, consider the following stored procedure. Use @@ERROR after every INSERT statement to see if the statement has completed successfully, shown in Listing 2-3.

LISTING 2-3 Procedure to show error checking with @@ERROR

Click here to view code image

```
CREATE PROCEDURE Examples.ErrorTesting_InsertTwo
AS
    SET NOCOUNT ON;
    INSERT INTO Examples.ErrorTesting(ErrorTestingId,
PositiveInteger)
    VALUES (3,3); --Succeeds

    IF @@ERROR <> 0
       BEGIN
            THROW 50000, 'First statement failed', 1;
            RETURN -1;
       END;

    INSERT INTO Examples.ErrorTesting(ErrorTestingId,
PositiveInteger)
    VALUES (4,-1); --Fail Constraint

    IF @@ERROR <> 0
       BEGIN
            THROW 50000, 'Second statement failed', 1;
            RETURN -1;
       END;

          INSERT INTO Examples.ErrorTesting(ErrorTestingId,
PositiveInteger)
          VALUES (5,1); --Will succeed if statement executes
          IF @@ERROR <> 0
       BEGIN
            THROW 50000, 'Third statement failed', 1;
            RETURN -1;
```

```
END;
```

Then run this procedure:

Click here to view code image

```
EXECUTE Examples.ErrorTesting_InsertTwo;
```

This gives you the following output (if following along, you can truncate the data if you want to run it multiple times):

Click here to view code image

```
Msg 547, Level 16, State 0, Procedure ErrorTesting_InsertTwo,
Line 12
The INSERT statement conflicted with the CHECK constraint
"CHKErrorTesting_PositiveInteger". The conflict occurred in
database
"ExamBook762Ch2", table "Examples.ErrorTesting", column
'PositiveInteger'.

Msg 50000, Level 16, State 1, Procedure ErrorTesting_InsertTwo,
Line 17
Second statement failed
```

You get both error messages, but you are able to stop the rest of the stored procedure execution, since the error message that was thrown was for the second insert statement.

Using TRY...CATCH

Using the TRY...CATCH construct is both far more powerful, and far easier to code with than using @@ERROR. The syntax is:

Click here to view code image

```
BEGIN TRY
    --Code you want to execute
END TRY
BEGIN CATCH
    --What to do if the code fails
END CATCH;
```

In the TRY section, you write your code as you normally would without error handling. If an error occurs, nothing is returned to the client immediately. Control is transferred to the CATCH section, and there you are able to decide what to do. You have access to information about the error through a set of system functions which are not cleared until the next error occurs, unlike @@ERROR. They are not scoped to the procedure or batch being run. If one procedure calls another, the called procedure can still see the error status information until an error occurs.

- **ERROR_NUMBER** Gives you the number of the error that caused you to be transferred to the **CATCH** section.
- **ERROR_MESSAGE** This is the text of the error message that was thrown.
- **ERROR_PROCEDURE** If the error occurred in a coded object, this contains the name of that object, otherwise it is NULL.
- **ERROR_LINE** This is the line of the batch or module where the error occurred.
- **ERROR_SEVERITY** The severity of the error. 16 is the normal error severity, higher are generally system errors.
- **ERROR_STATE** The extended error state value that an error message can include.

Note More on error states

For more information about error severities, the following article on MSDN has an explanation of all of them: *https://msdn.microsoft.com/en-us/library/ms164086.aspx*. For more information about error states, consult *https://msdn.microsoft.com/en-us/library/ms180031.aspx*.

In Listing 2-4 we have written a much simpler bit of code. None of it should be too surprising, but note the THROW statement to end the CATCH section. Using THROW in this manner works in a CATCH block to send the error message to the caller that caused the CATCH to be called (typically referred to as *rethrowing* an error). This allows you to write code to deal with the error, undo changes, log the error message, etc., and then present the error message to the client as it would have looked without being caught.

LISTING 2-4 Procedure to show error checking with TRY...CATCH

Click here to view code image

```
ALTER PROCEDURE Examples.ErrorTesting_InsertTwo
AS
    SET NOCOUNT ON;
    DECLARE @Location nvarchar(30);

    BEGIN TRY
        SET @Location = 'First statement';
        INSERT INTO Examples.ErrorTesting(ErrorTestingId,
PositiveInteger)
        VALUES (6,3); --Succeeds

        SET @Location = 'Second statement';
        INSERT INTO Examples.ErrorTesting(ErrorTestingId,
PositiveInteger)
```

```
        VALUES (7,-1); --Fail Constraint

        SET @Location = 'First statement';
        INSERT INTO Examples.ErrorTesting(ErrorTestingId,
PositiveInteger)
        VALUES (8,1); --Will succeed if statement executes
    END TRY
    BEGIN CATCH
        SELECT ERROR_PROCEDURE() AS ErrorProcedure, @Location
AS ErrorLocation
        SELECT ERROR_MESSAGE() as ErrorMessage;
        SELECT ERROR_NUMBER() AS ErrorNumber, ERROR_SEVERITY()
as ErrorSeverity,
               ERROR_LINE() As ErrorLine;

               THROW;

           END CATCH;
```

The major part of the configuration is just coding the TRY...CATCH blocks. The only manual bits of code you may optionally wish to use in the TRY section is to save off the location into a variable. You can use the line number, but this is quite confusing as errors can be bubbled up from other stored procedures, triggers, etc.

Run this procedure:

Click here to view code image

```
EXECUTE Examples.ErrorTesting_InsertTwo;
```

The output tells you all of the details, including the data we selected out, and the error message that was rethrown.

Click here to view code image

```
ErrorProcedure                  ErrorLocation
------------------------------- -----------------------------
ErrorTesting_InsertTwo          Second statement

ErrorMessage
------------------------------------------------------------------
----------------------
The INSERT statement conflicted with the CHECK constraint
"CHKErrorTesting_PositiveInteger". The conflict occurred in
database "ExamBook762Ch2",
table "Examples.ErrorTesting", column 'PositiveInteger'.

ErrorNumber ErrorSeverity ErrorLine
----------- ------------- -----------
```

```
547         16            12

Msg 547, Level 16, State 0, Procedure ErrorTesting_InsertTwo,
Line 12
The INSERT statement conflicted with the CHECK constraint
"CHKErrorTesting_PositiveInteger". The conflict occurred in
database "ExamBook762Ch2",
table "Examples.ErrorTesting", column 'PositiveInteger'.
```

Transaction Control Logic in Your Error Handling

In this final section on error handling, we combine all of the concepts we have discussed, along with an initial review of transactions (Chapter 3 will go much deeper into the various forms of transaction handling that you may need, but transactions are essential to a discussion of transactions). Every statement in SQL Server, DDL and DML alike, are performed as a transaction. By default, they are all considered *autocommit* transactions. When you want to run multiple statements together, making sure that they all complete, you use *explicit transactions*. Do this by using the BEGIN TRANSACTION statement to start a transaction, and COMMIT TRANSACTION to save the changes, or ROLLBACK TRANSACTION to undo the changes that have been made.

Transactions can be nested, as in:

```
BEGIN TRANSACTION;
BEGIN TRANSACTION;
```

You can tell how many transactions have been nested by using the @@TRANCOUNT system function.

```
SELECT @@TRANCOUNT
```

After running the two BEGIN TRANSACTION statements, the result of this query is 2. To save the changes made after the BEGIN TRANSACTION statements, you need an equal number of COMMIT TRANSACTION calls to save the changes. While syntactically there are two nested transactions, there is technically just a single transaction internally. To undo your changes, you only need a solitary ROLLBACK TRANSACTION call to undo the changes.

For example, run the following batch of statements:

Click here to view code image

```
BEGIN TRANSACTION;
INSERT INTO Examples.ErrorTesting(ErrorTestingId,
PositiveInteger)
VALUES (9,1);

BEGIN TRANSACTION;
SELECT * FROM Examples.ErrorTesting WHERE ErrorTestingId = 9;
```

```
ROLLBACK TRANSACTION;
SELECT * FROM Examples.ErrorTesting WHERE ErrorTestingId = 9;
```

The statement succeeds, as no error is returned, and then the first SELECT statement returns data, but the second does not. One final system function that we need for the error handler is XACT_STATE(). You use this function to determine the current status of a transaction if one is in effect. There are three possible values: 1-There is an active transaction that can be committed; 0-There is no active transaction; -1-There is an active transaction that cannot be committed, also referred to as an *uncommitable transaction*, or a *doomed* transaction. An uncommitable transaction is caused by a few rare situations that can occur in complex code such as using XACT_ABORT with an error handler. (XACT_ABORT is a SET options that ends the batch on a transaction that we show later in this section. It is typically not used with any other error handling).

In the next three code listings, there are three possible transaction and error handling schemes that make sure that either all statements succeed, or they all fail. Our scenario uses the following two tables (note the CHECK constraint on the CompanyName column so we can force an error on the second table):

Click here to view code image

```
CREATE TABLE Examples.Worker
(
    WorkerId int NOT NULL IDENTITY(1,1) CONSTRAINT PKWorker
PRIMARY KEY,
    WorkerName nvarchar(50) NOT NULL CONSTRAINT AKWorker UNIQUE
);
CREATE TABLE Examples.WorkerAssignment
(
    WorkerAssignmentId int IDENTITY(1,1) CONSTRAINT
PKWorkerAssignment PRIMARY KEY,
    WorkerId int NOT NULL,
    CompanyName nvarchar(50) NOT NULL
       CONSTRAINT CHKWorkerAssignment_CompanyName
           CHECK (CompanyName <> 'Contoso, Ltd.'),
    CONSTRAINT AKWorkerAssignment UNIQUE (WorkerId,
CompanyName)
);
```

To keep the processing very simple, the requirements for the STORED PROCEDURE object we are creating is to create one Worker row and one WorkerAssignment row in a single call to the STORED PROCEDURE. Hence, the basic part of the code is to perform:

Click here to view code image

```
INSERT INTO Examples.Worker...
INSERT INTO Examples.WorkerAssignment...
```

If either of the statements fails, the goal is to capture the error, return the error telling the user where in the code the error occurred, and end the batch. In Listing 2.5, we start by implementing this with a TRY...CATCH construct. The code include comments that explain anything new, and to clarify what is being accomplished.

LISTING 2-5 Procedure to show realistic error checking with TRY...CATCH

Click here to view code image

```
CREATE PROCEDURE Examples.Worker_AddWithAssignment
    @WorkerName nvarchar(50),
    @CompanyName nvarchar(50)
AS
    SET NOCOUNT ON;
    --do any non-data testing before starting the transaction
    IF @WorkerName IS NULL or @CompanyName IS NULL
        THROW 50000,'Both parameters must be not null',1;

    DECLARE @Location nvarchar(30), @NewWorkerId int;
    BEGIN TRY
        BEGIN TRANSACTION;

        SET @Location = 'Creating Worker Row';
        INSERT INTO Examples.Worker(WorkerName)
        VALUES (@WorkerName);

        SELECT @NewWorkerId = SCOPE_IDENTITY(),
               @Location = 'Creating WorkAssignment Row';

        INSERT INTO Examples.WorkerAssignment(WorkerId,
CompanyName)
        VALUES (@NewWorkerId, @CompanyName);

        COMMIT TRANSACTION;
    END TRY
    BEGIN CATCH
        --at the end of the call, we want the transaction
rolled back
        --rollback the transaction first, so it definitely
occurs as the THROW
        --statement would keep it from happening.
        IF XACT_STATE() <> 0 --if there is a transaction in
effect
                             --commitable or not
            ROLLBACK TRANSACTION;
```

```
        --format a message that tells the error and then THROW
it.
        DECLARE @ErrorMessage nvarchar(4000);
        SET @ErrorMessage = CONCAT('Error occurred during:
''',@Location,'''',
                                   ' System Error: ',
                                   ERROR_NUMBER(),':',ERROR_MES
        THROW 50000, @ErrorMessage, 1;
    END CATCH;
```

Showing how the code works, first try NULL parameter values.

Click here to view code image

```
EXEC Examples.Worker_AddWithAssignment @WorkerName = NULL,
@CompanyName = NULL;
```

This returns the following error, which happens even before the explicit transaction is started, which would have given an error. In some cases, it can be advantageous to check for certain types of errors before modifying data.

Click here to view code image

```
Msg 50000, Level 16, State 1, Procedure
Worker_AddWithAssignment, Line 7
Both parameters must be not null
```

Next, insert a set of rows that succeed:

Click here to view code image

```
EXEC Examples.Worker_AddWithAssignment
                          @WorkerName='David So',
@CompanyName='Margie''s Travel';
```

You can see what happens when there is an error by running the following statement with the same @WorkerName parameter value:

Click here to view code image

```
EXEC Examples.Worker_AddWithAssignment
                          @WorkerName='David So',
@CompanyName='Margie''s Travel';
```

This results in the following error:

Click here to view code image

```
Msg 50000, Level 16, State 1, Procedure
Worker_AddWithAssignment, Line 38
Error occurred during: 'Creating Worker Row' System Error:
2627:Violation of UNIQUE KEY
```

```
constraint 'AKWorker'. Cannot insert duplicate key in object
'Examples.Worker'. The
duplicate key value is (David So).
```

Now, show that a failure when an error occurs with the second table being referenced:

Click here to view code image

```
EXEC Examples.Worker_AddWithAssignment
                          @WorkerName='Ian Palangio',
@CompanyName='Contoso, Ltd.';
```

This returns the following:

Click here to view code image

```
Msg 50000, Level 16, State 1, Procedure
Worker_AddWithAssignment, Line 38
Error occurred during: 'Creating WorkAssignment Row' System
Error: 547:The INSERT
statement conflicted with the CHECK constraint
"CHKWorkerAssignment_CompanyName". The
conflict occurred in database "ExamBook762Ch2", table
"Examples.WorkerAssignment",
column 'CompanyName'.
```

Then you can make sure it works by changing the @CompanyName parameter value.

Click here to view code image

```
EXEC Examples.Worker_AddWithAssignment
                       @WorkerName='Ian Palangio',
@CompanyName='Humongous Insurance';
```

In Listing 2-6, we have the same goals for the stored procedure, but instead use @@error to determine if an error has occurred.

LISTING 2-6 Procedure to show realistic error checking with @@ERROR

Click here to view code image

```
ALTER PROCEDURE Examples.Worker_AddWithAssignment
    @WorkerName nvarchar(50),
    @CompanyName nvarchar(50)
AS
    SET NOCOUNT ON;
    DECLARE @NewWorkerId int;
    --still check the parameter values first
    IF @WorkerName IS NULL or @CompanyName IS NULL
        THROW 50000,'Both parameters must be not null',1;
```

```
    --Start a transaction
    BEGIN TRANSACTION
    INSERT INTO Examples.Worker(WorkerName)
    VALUES (@WorkerName);
    --check the value of the @@error system function
    IF @@ERROR <> 0
      BEGIN
        --rollback the transaction before the THROW (or RETURN
if using), because
        --otherwise the THROW will end the batch and
transaction stay open
        ROLLBACK TRANSACTION;
        THROW 50000,'Error occurred inserting data into
Examples.Worker table',1;
      END;
    SELECT @NewWorkerId = SCOPE_IDENTITY()

    INSERT INTO Examples.WorkerAssignment(WorkerId,
CompanyName)
    VALUES (@NewWorkerId, @CompanyName);
     IF @@ERROR <> 0
      BEGIN
        ROLLBACK TRANSACTION;
        THROW 50000,
          'Error occurred inserting data into
Examples.WorkerAssignment table',1;
      END;
    --if you get this far in the batch, you can commit the
transaction
    COMMIT TRANSACTION;
```

Now, run the following two commands. GO is a batch separator that splits the two executions into two independent communications with the server, so the second runs no matter what happens with the first, unless something drastic happens and the connection to the server is lost:

Click here to view code image

```
EXEC Examples.Worker_AddWithAssignment @WorkerName='Seth
Grossman', @
CompanyName='Margie''s Travel';

GO
--Cause an error due to duplicating all of the data from
previous call
EXEC Examples.Worker_AddWithAssignment @WorkerName='Seth
Grossman', @
```

```
CompanyName='Margie''s Travel';
```

The second call returns 2 errors, the first being the error from the command, and the second coming from the THROW statement:

Click here to view code image

```
Msg 2627, Level 14, State 1, Procedure
Worker_AddWithAssignment, Line 14
Violation of UNIQUE KEY constraint 'AKWorker'. Cannot insert
duplicate key in object
'Examples.Worker'. The duplicate key value is (Seth Grossman).

Msg 50000, Level 16, State 1, Procedure
Worker_AddWithAssignment, Line 21
Error occurred inserting data into Examples.Worker table
```

Finally, in Listing 2-7, we demonstrate the final method of error-handling in a stored procedure using XACT_ABORT ON. The way this works is that when running, if an error occurs, the batch stops and the transaction is stopped. It is effective, but gives you no real control over what happens in an error. Since the batch ends immediately, to know what statement you are executing you need to print messages constantly. Hence this method is more used for system tasks, but it is an effective tool for dealing with rolling back a transaction on an error.

LISTING 2-7 Procedure to show stopping transaction and batch using SET XACT_ABORT ON

Click here to view code image

```
ALTER PROCEDURE Examples.Worker_AddWithAssignment
    @WorkerName nvarchar(50),
    @CompanyName nvarchar(50)
AS
    SET NOCOUNT ON;
    --will cause batch to end on any error
    SET XACT_ABORT ON;

    DECLARE @NewWorkerId int;

    --Same parameter check as other cases
    IF @WorkerName IS NULL or @CompanyName IS NULL
        THROW 50000,'Both parameters must be not null',1;

    --start the transaction
    BEGIN TRANSACTION;
    --  Execute the code as normal
```

```
    INSERT INTO Examples.Worker(WorkerName)
    VALUES (@WorkerName);

    SELECT @NewWorkerId = SCOPE_IDENTITY()

    INSERT INTO Examples.WorkerAssignment(WorkerId,
CompanyName)
    VALUES (@NewWorkerId, @CompanyName);

    COMMIT TRANSACTION;
```

Now all you get is just the system error message returned, as you see from the following:

Click here to view code image

```
EXEC Examples.Worker_AddWithAssignment
            @WorkerName='Stig Panduro', @CompanyName='Margie''s
Travel';

GO
--Cause an error due to duplicating all of the data from
previous call
EXEC Examples.Worker_AddWithAssignment
            @WorkerName='Stig Panduro', @CompanyName='Margie''s
Travel';
```

This returns:

Click here to view code image

```
Msg 2627, Level 14, State 1, Procedure
Worker_AddWithAssignment, Line 12
Violation of UNIQUE KEY constraint 'AKWorker'. Cannot insert
duplicate key in object
'Examples.Worker'. The duplicate key value is (Stig Panduro).
```

A consideration to note when building your error handling is that a STORED PROCEDURE cannot change the transaction count from when it starts to when it finishes. For example, consider the following STORED PROCEDURE object:

Click here to view code image

```
CREATE PROCEDURE ChangeTransactionLevel
AS
    BEGIN TRANSACTION;
    ROLLBACK TRANSACTION;
```

If you perform this outside of the context of a transaction, everything works fine with no errors. But if place this in a transaction:

```
BEGIN TRANSACTION;
EXEC ChangeTransactionLevel;
ROLLBACK TRANSACTION;
```

You receive the following error messages:

Click here to view code image

```
Msg 266, Level 16, State 2, Procedure ChangeTransactionLevel,
Line 0
Transaction count after EXECUTE indicates a mismatching number
of BEGIN and COMMIT
statements. Previous count = 1, current count = 0.
Msg 3903, Level 16, State 1, Line 1434
The ROLLBACK TRANSACTION request has no corresponding BEGIN
TRANSACTION.
```

The second message is not critical, unless you expect to be in a transaction at the time and keep modifying data or structure. There are two very standard steps to mitigate this issue. First, as we have done in the TRY...CATCH example previously, end your error handler with a THROW statement, which ends the batch:

Click here to view code image

```
ALTER PROCEDURE ChangeTransactionLevel
AS
    BEGIN TRANSACTION;
    ROLLBACK TRANSACTION;
    THROW 50000,'Error After Rollback',1;
```

Now, perform this in the same transaction:

```
BEGIN TRANSACTION;
EXEC ChangeTransactionLevel;
ROLLBACK TRANSACTION;
```

You see just the one error message:

Click here to view code image

```
Msg 50000, Level 16, State 1, Procedure ChangeTransactionLevel,
Line 5
Error After Rollback
```

The alternative is to use what are called *savepoints*. Savepoints allow you to roll back part of a transaction, and is covered in Skill 3.1. Prior to THROW, savepoints were common for use in error handling, but now are used primarily for more specific purposes. Generally, using THROW after making sure that you have ended all transactions is the cleanest method of execution.

For the most part, anytime you have complex stored procedures, you should be using

TRY...CATCH and THROW. These commands simplify dealing with errors. This is particularly true when you have a procedure that is called by another procedure that can cause errors (which is pretty much any statement in Transact-SQL).

For example, consider the following procedure that is used to call the example procedure we just created that changed the transaction level. Use the same error handler that we used earlier in the TRY...CATCH example, as seen in Listing 2-8.

LISTING 2-8 Procedure to what happens when you change the transaction level in a procedure

Click here to view code image

```
ALTER PROCEDURE dbo.CallChangeTransactionLevel
AS
    BEGIN TRY
        BEGIN TRANSACTION

        DECLARE @Location nvarchar(30) = 'Execute Procedure';
        EXECUTE ChangeTransactionLevel; --This will cause an
error by design

        COMMIT TRANSACTION
    END TRY
    BEGIN CATCH
        IF XACT_STATE() <> 0
            ROLLBACK;
        DECLARE @ErrorMessage nvarchar(4000)
        SET @ErrorMessage = CONCAT('Error occurred during:
''',@Location,'''',
                                   ' System Error:
',ERROR_NUMBER(),':',
                                   ERROR_MESSAGE());
        THROW 50000, @ErrorMessage, 1;
    END CATCH;
```

Now, perform the calling procedure:

Click here to view code image

```
EXECUTE dbo.CallChangeTransactionLevel;
```

This gives you the error from the procedure that said Error After Rollback:

Click here to view code image

```
Msg 50000, Level 16, State 1, Procedure
CallChangeTransactionLevel, Line 19
```

```
Error occurred during: 'Execute Procedure' System Error:
50000:Error After Rollback
```

Now you can see the error from all of the calling procedures, in a stack as each CATCH block appends the message from the previous call. You could add an ERROR_PROCEDURE() function call to the CONCAT expression for the error message to make the entire stack more obvious, or even use RAISERROR to return a message at each level, and a THROW command to stop the batch when you have reached the top level. You can tell this using the @@nestlevel system function, but usually this is enough of error handler for the typical need, particularly keeping it simple enough for this review of error handling.

Need More Review? The complexities of error handling

To be certain, error handling is a complex topic that cannot be given a complete review in a book of this size and purpose. One of the best resources on Error and Transaction Handling is from Erland Sommarskog, here on his website: *http://sommarskog.se/error_handling/Part1.html*.

Skill 2.3 Create triggers and user-defined functions

In this final skill of the chapter, we cover two very different, and somewhat lesser-used features of Transact-SQL, with very little overlap between them.

First we look at TRIGGER objects, which in itself is a very large topic, particularly because there are three different types of triggers that you can work with, but equally because they are quite a bit more complex than STORED PROCEDURE objects. You use them to react to some event, either a DML operation such as an INSERT, UPDATE or DELETE statement execution; someone changing a setting or object on the server; or even someone logging into a server.

Then we cover the details of User-Defined Functions (or UDFs) that allow you to create code that is called in the same way that a system function might. UDFs are very powerful tools, but are also very dangerous for performance when used poorly.

Note TRIGGER objects

This chapter focuses on interpreted Transact-SQL objects. You can create TRIGGER objects in managed code. Skill 3.4 highlights the differences between these solutions and the memory-optimized tables and natively-compiled modules.

This section covers how to:

- Design trigger logic based on business requirements
- Determine when to use Data Manipulation Language (DML) triggers, Data Definition Language (DDL) triggers, or logon triggers
- Recognize results based on execution of AFTER or INSTEAD OF triggers
- Design scalar-valued and table-valued user-defined functions based on business requirements
- Identify differences between deterministic and non-deterministic functions

Design trigger logic based on business requirements

Triggers are coded objects, similar to stored procedures, which allow you to run code in response to events that occur in SQL Server. The most common types of triggers fire on a DML operation such as an INSERT, UPDATE or DELETE statement execution. Additionally, there are triggers that fire when someone changes something in SQL Server (DDL triggers), a setting or object on the server, or even someone logging into a server (login triggers).

In this section, we focus on DML triggers, because they are the ones that are generally

used for business requirements. In the next section we review the other types of triggers, which are mostly focused on administration needs. DML TRIGGER objects are schema owned, database contained objects, like STORED PROCEDURE, VIEW, and CONSTRAINT objects, so their names must not collide with other objects in the database.

DML TRIGGER objects are typically used for a couple of purposes that are hard to do in the declarative configuration of tables. The logic often could be placed in a STORED PROCEDURE object, but by using a TRIGGER object, you can limit duplication of code:

- **Complex data integrity** CHECK constraints can only see data in the same row. If you need to check data across multiple rows, only triggers can do this automatically.
- **Running code in response to some action** For example, if an order comes in past a threshold (like a $3,000,000 order for lattes from Fourth Coffee), you could write a row to a table to have the row checked for validity.
- **Ensuring columnar data is modified** If you want to make sure that data is modified, like a column that tells you when a row was last modified, triggers can ensure that the user does not put in invalid data.
- **Making a view editable** If a VIEW references more than one table, it becomes complicated to modify it using simple DML operations.

There are two different types of DML Trigger objects that work for INSERT, UPDATE, and DELETE operations that you should familiarize yourself with:

- **AFTER** These triggers perform after a DML operation. They are typically used for doing data validations, as you can see the data as it is after the operation has occurred.
- **INSTEAD OF** These triggers perform instead of the DML operation, so if you want the operation to occur, you need repeat the DML in the code.

The following subsections contain an example of each of the scenarios that are listed, along with some commentary on the major pitfalls you encounter. This is not by any means an exhaustive list of ways that triggers can be used, but a simple overview of how they are be created to implement given needs.

> **Need More Review? More on the CREATE TRIGGER statement**
>
> Triggers are not a simple topic that we can cover in any real depth, not even to the light depth we have reviewed other topics. If you need to study more about the many details of writing triggers, a good start is the MSDN page on CREATE TRIGGER here: *https://msdn.microsoft.com/en-us/library/ms189799.aspx*.

Complex data integrity

The automatic data-integrity enforcement we have covered so far has been of the

declarative variety. You state a predicate in the form of a column declaration/data type and possibly a constraint. A major limitation of constraints is how much they can see. For example, CHECK constraints can only see data in the same row. FOREIGN KEY constraints can only see the current row and see if another row exists.

While this covers a great percentage of needs, if you need to check data across a group of rows, triggers can do this (technically CHECK constraints can use a USER DEFINED FUNCTION, but TRIGGER objects are considered the best method). For this example, consider you have the following table (just including columns that we need for the example):

Click here to view code image

```
CREATE TABLE Examples.AccountContact
(
     AccountContactId int NOT NULL CONSTRAINT PKAccountContact
PRIMARY KEY,
     AccountId        char(4) NOT NULL,
     PrimaryContactFlag bit NOT NULL
);
```

You are given the business requirement to ensure there is always one primary contact for an account, if a contact does exist. A first step is to identify the query that shows you rows that do not match this rule. In this case:

Click here to view code image

```
SELECT AccountId, SUM(CASE WHEN PrimaryContactFlag = 1 THEN 1
ELSE 0 END)
FROM   Examples.AccountContact
GROUP BY AccountId
HAVING SUM(CASE WHEN PrimaryContactFlag = 1 THEN 1 ELSE 0 END)
<> 1;
```

If that query returns data, then you know something is wrong. This query is the basis of the data check. We can put this into a trigger as seen in Listing 2-9, which checks on any INSERT or UPDATE operation.

LISTING 2-9 Trigger to stop multiple PrimaryContacts for an Account during an UPDATE or INSERT operation

Click here to view code image

```
CREATE TRIGGER Examples.AccountContact_TriggerAfterInsertUpdate
ON Examples.AccountContact
AFTER INSERT, UPDATE AS
BEGIN
  SET NOCOUNT ON;
```

```
  SET ROWCOUNT 0; --in case the client has modified the
rowcount
  BEGIN TRY
  --check to see if data is returned by the query from
previously
  IF EXISTS ( SELECT AccountId
              FROM   Examples.AccountContact
                     --correlates the changed rows in inserted
to the other rows
                     --for the account, so we can check if the
rows have changed
              WHERE  EXISTS (SELECT *
                             FROM   inserted
                             WHERE  inserted.AccountId =
                                                AccountContact

                             UNION ALL
                             SELECT *
                             FROM   deleted
                             WHERE  deleted.AccountId =
                                                AccountContact
              GROUP BY AccountId
              HAVING SUM(CASE WHEN PrimaryContactFlag = 1 then
1 ELSE 0 END) <> 1)

          THROW  50000, 'Account(s) do not have only one
primary contact.', 1;
   END TRY
   BEGIN CATCH
        IF XACT_STATE() <> 0
            ROLLBACK TRANSACTION;
        THROW;
     END CATCH
END;
```

Note Multiple `AFTER` triggers

It is possible to have multiple AFTER triggers on the same operation. However, you have minimal control over the order in which they run. For more information, see the MSDN article on sp_settriggerorder system stored procedure: *https://msdn.microsoft.com/en-us/library/ms186762.aspx*.

For the most part, this is pretty straightforward. We give the TRIGGER a name, tell it the TABLE it is for, and then that this fires AFTER an INSERT or UPDATE operation. Then it is just like a stored procedure for the most part. There are two virtual tables inserted and

deleted that instantiated when the trigger performs. Inserted shows you how the data looks after the operation, and deleted shows you how the data looks before the operation. Both tables only have data during an UPDATE operation, showing you the before and the after versions. In some cases, both tables are empty, such as an UPDATE statement where the WHERE clause matched no rows, or a MERGE statement where one of the operations had no matches.

The most important part of writing such a trigger is that you must be prepared for more than one row to be modified. In the EXISTS block, we have the query we started from, and added the WHERE EXISTS condition to limit the scope of the query to just rows that have been modified based on their AccountId. You have to use both inserted and deleted table rows because there is nothing stopping the AccountId from changing:

Click here to view code image

```
SELECT AccountId
FROM   Examples.AccountContact
--correlates the changed rows in inserted to the other rows
--for the account, so we can check if the rows have changed
-------------------------------------------------------------
WHERE  EXISTS (SELECT *
               FROM   inserted
               WHERE  inserted.AccountId =
AccountContact.AccountId
               UNION ALL
               SELECT *
               FROM   deleted
               WHERE  deleted.AccountId =
AccountContact.AccountId)
-------------------------------------------------------------
GROUP BY AccountId
HAVING SUM(CASE WHEN PrimaryContactFlag = 1 then 1 ELSE 0 END;
```

Many trigger writers make the mistake of writing variable declaration statements to grab values from the inserted/deleted virtual table like the following:

Click here to view code image

```
SELECT @AccountId = AccountId FROM inserted;
```

Using that AccountId to check for issues misses all but the one row. You must, though, test all of the cases with single and multiple rows. For this trigger, consider running at least the following simple tests:

Click here to view code image

```
--Success, 1 row
INSERT INTO Examples.AccountContact(AccountContactId,
AccountId, PrimaryContactFlag)
```

```
VALUES (1,1,1);
--Success, two rows
INSERT INTO Examples.AccountContact(AccountContactId,
AccountId, PrimaryContactFlag)
VALUES (2,2,1),(3,3,1);
--Two rows, same account
INSERT INTO Examples.AccountContact(AccountContactId,
AccountId, PrimaryContactFlag)
VALUES (4,4,1),(5,4,0);
--Invalid, two accounts with primary
INSERT INTO Examples.AccountContact(AccountContactId,
AccountId, PrimaryContactFlag)
VALUES (6,5,1),(7,5,1);
```

This returns:

Click here to view code image

```
Msg 50000, Level 16, State 1, Procedure
AccountContact_TriggerAfterInsert, Line 29
One or more Accounts does not have one and only one primary
contact.
```

Then, without showing messages:

Click here to view code image

```
--Invalid, no primary
INSERT INTO Examples.AccountContact(AccountContactId,
AccountId, PrimaryContactFlag)
VALUES (8,6,0),(9,6,0);
--Won't work, because AccountId is new, and this row is not
primary
UPDATE Examples.AccountContact
SET    AccountId = 6
WHERE  AccountContactId = 5;
```

Triggers can be tricky to get right, and logically tracing through the code and testing is important. On the exam, if presented with a trigger, it is important to be very careful to be able to trace through the single-row and multi-row operations that are needed.

The requirements for our problem stated that we are to make sure every account has a primary contact, if a contact exists. As it stands now, while the user can't create or modify rows to violate the requirement, a user can delete the primary row. So, creating a DELETE TRIGGER works very similar to the INSERT/UPDATE one, except now you use the deleted virtual table, as you can see in Listing 2-10.

LISTING 2-10 Trigger to stop multiple PrimaryContacts for an Account during a DELETE operation

Click here to view code image

```
CREATE TRIGGER Examples.AccountContact_TriggerAfterDelete
ON Examples.AccountContact
AFTER DELETE AS
BEGIN
   SET NOCOUNT ON;
   SET ROWCOUNT 0; --in case the client has modified the
rowcount
   BEGIN TRY
   IF EXISTS ( SELECT AccountId
                FROM   Examples.AccountContact
                WHERE  EXISTS (SELECT *
                               FROM   deleted
                               WHERE  deleted.AccountId =
                                   AccountContact.Accoun
            GROUP BY AccountId
            HAVING SUM(CASE WHEN PrimaryContactFlag = 1 then 1
ELSE 0 END) > 1)
       THROW  50000, 'One or more Accounts did not have one
primary contact.', 1;
   END TRY
   BEGIN CATCH
       IF XACT_STATE() <> 0
           ROLLBACK TRANSACTION;
       THROW;
    END CATCH;
END;
```

The basic structure of the TRIGGER for data integrity is to see if there is an issue, either by checking data in the inserted/deleted tables, the primary table, or any tables, and if there is an issue, run a THROW statement and let the ROLLBACK TRANSACTION undo the statement's effect and any other activity done within the context of the transaction. If the caller has a TRY...CATCH block, they get the error captured. If not, the batch ends due to the THROW statement. If you use RAISERROR, things are trickier because the batch continues after the transaction ends.

Running code in response to some action

There are many situations where a modification is made to a row that you want to affect a change of some sort in another table. For example, consider a table that captures promises to a charity. The table might look something like the following, including only columns pertinent to the example:

Click here to view code image

```
CREATE TABLE Examples.Promise
(
    PromiseId int NOT NULL CONSTRAINT PKPromise PRIMARY KEY,
    PromiseAmount money NOT NULL
);
```

No matter the charity, there are a few levels of promises that can be received. For simplicity, let's define two: Normal and Extranormal. A Normal promise is in a typical range that a person promises if they are normal and sincere. Extranormal promises are outside of the Normal and need verification. Extranormal promises for this scenario are those over $10,000.00. The requirements are to create a log of promises to verify when rows are created or updated.

So, you design a table that has the Promise rows to be verified, which looks like this (without the details of the row being verified):

Click here to view code image

```
CREATE TABLE Examples.VerifyPromise
(
    VerifyPromiseId int NOT NULL CONSTRAINT PKVerifyPromise
PRIMARY KEY,
    PromiseId int NOT NULL CONSTRAINT AKVerifyPromise UNIQUE
                --FK not included for simplicity
);
```

In Listing 2-11 the TRIGGER object fulfills this requirement.

LISTING 2-11 Trigger to create rows in another table

Click here to view code image

```
CREATE TRIGGER Examples.Promise_TriggerInsertUpdate
ON Examples.Promise
AFTER INSERT, UPDATE AS
BEGIN
   SET NOCOUNT ON;
   SET ROWCOUNT 0; --in case the client has modified the
rowcount
   BEGIN TRY
       INSERT INTO Examples.VerifyPromise(PromiseId)
       SELECT PromiseId
       FROM   inserted
       WHERE  PromiseAmount > 10000.00
         AND  NOT EXISTS (SELECT * --keep from inserting
duplicates
                          FROM   VerifyPromise
                          WHERE  VerifyPromise.PromiseId =
```

```
inserted.PromiseId)
   END TRY
   BEGIN CATCH
       IF XACT_STATE() <> 0
           ROLLBACK TRANSACTION;
       THROW; --will halt the batch or be caught by the
caller's catch block
   END CATCH
END;
```

The biggest thing to note in this TRIGGER is how you need to do very little error handling. Just use the TRY...CATCH block to see if there is an error with the statement, and if so, run the ROLLBACK TRANSACTION statement that rethrows the error message.

> **Note Triggers modifying data**
>
> When a trigger modifies data in the same or different table, there can be triggers that also get fired. Discussion of this is beyond the scope of this review, but there is an article on MSDN that covers this in detail: *https://msdn.microsoft.com/en-us/library/ms190739.aspx*.

Ensuring columnar data is modified

In this example, we make use of INSTEAD OF TRIGGER objects, which are excellent tools for making sure some operation occurs in a statement. For example, if you want to make sure that a column tells you when a row was last modified, an INSTEAD OF TRIGGER object can be used to determine if the user inputs data that does not make sense.

Consider the following TABLE:

Click here to view code image

```
CREATE TABLE Examples.Lamp
(
    LampId       int IDENTITY(1,1) CONSTRAINT PKLamp PRIMARY
KEY,
    Value          varchar(10) NOT NULL,
    RowCreatedTime datetime2(0) NOT NULL
        CONSTRAINT DFLTLamp_RowCreatedTime
DEFAULT(SYSDATETIME()),
    RowLastModifiedTime datetime2(0) NOT NULL
        CONSTRAINT DFLTLamp_RowLastModifiedTime
DEFAULT(SYSDATETIME())
);
```

While we specified a DEFAULT constraint, the user can put anything at all in the table. Instead, let's use two TRIGGER objects. The first is an INSTEAD OF INSERT TRIGGER

object as seen in Listing 2-12.

LISTING 2-12 INSTEAD OF TRIGGER to automatically set RowCreated and RowLastModified time columns

Click here to view code image

```
CREATE TRIGGER Examples.Lamp_TriggerInsteadOfInsert
ON Examples.Lamp
INSTEAD OF INSERT AS
BEGIN
   SET NOCOUNT ON;
   SET ROWCOUNT 0; --in case the client has modified the
rowcount
   BEGIN TRY
        --skip columns to automatically set
        INSERT INTO Examples.Lamp( Value)
        SELECT Value
        FROM   inserted
   END TRY
   BEGIN CATCH
       IF XACT_STATE() <> 0
           ROLLBACK TRANSACTION;
       THROW; --will halt the batch or be caught by the
caller's catch block
   END CATCH
END;
```

Note More on INSTEAD OF triggers

You can only have one INSTEAD OF trigger per operation on a table. While you can have one INSTEAD OF TRIGGER object that does multiple operations, like INSERT, UPDATE, and DELETE, it is not typically as useful as it can be for AFTER TRIGGER objects. One use case is to make a trigger not do the actual operation.

This trigger is very similar to the one in previous sections. The biggest difference is that the INSERT statement is doing the operation that the user expected it was doing, but skipping the columns that use a defaulted value. You can use this step in the process to do any formatting that you don't want the user to have control over. After inserting a row and viewing it, you see the following:

Click here to view code image

```
INSERT INTO Examples.Lamp(Value, RowCreatedTime,
```

```
   RowLastModifiedTime)
   VALUES ('Original','1900-01-01','1900-01-01');

   SELECT *
   FROM   Examples.Lamp;
```

Here are the two columns:

[Click here to view code image](#)

```
   LampId      Value      RowCreatedTime              RowLastModif
   ----------- ---------- --------------------------- ------------
   ---------------
   1           Original   2016-09-20 21:03:54         2016-09-20
   21:03:54
```

Next, create the INSTEAD OF UPDATE TRIGGER that makes sure that the RowLastModifiedTime is modified, and the RowCreatedTime is never modified.

[Click here to view code image](#)

```
   CREATE TRIGGER Examples.Lamp_TriggerInsteadOfUpdate
   ON Examples.Lamp
   INSTEAD OF UPDATE AS
   BEGIN
      SET NOCOUNT ON;
      SET ROWCOUNT 0; --in case the client has modified the
   rowcount
      BEGIN TRY
           UPDATE Lamp
           SET    Value = inserted.Value,
                  RowLastModifiedTime = DEFAULT --use default
   constraint
           FROM   Examples.Lamp
                    JOIN inserted
                       ON Lamp.LampId = inserted.LampId;
      END TRY
      BEGIN CATCH
          IF XACT_STATE() <> 0
              ROLLBACK TRANSACTION;
          THROW; --will halt the batch or be caught by the
   caller's catch block
      END CATCH;
   END;
```

This is similar to the INSERT trigger, but we do the UPDATE instead of the INSERT. This time, skip the RowCreatedTime column because the time the row was created doesn't change, but the modified time does. Update and view the row previously created:

[Click here to view code image](#)

```
UPDATE Examples.Lamp
SET    Value = 'Modified',
       RowCreatedTime = '1900-01-01',
       RowLastModifiedTime = '1900-01-01'
WHERE LampId = 1;

SELECT *
FROM   Examples.Lamp;
```

The RowLastModifiedTime is different than the first call, and different now from the RowCreatedTime:

Click here to view code image

```
LampId      Value      RowCreatedTime              RowLastModif
----------- ---------- --------------------------- ------------
---------------
1           Modified   2016-09-20 21:07:07         2016-09-20
21:10:26
```

Making any view modifiable using INSTEAD OF triggers

A final example of DML triggers is to apply an INSTEAD OF TRIGGER to a VIEW object, making it editable. It can even be editable if the view isn't based on a table. No matter what the data that is returned from a SELECT statement on the view, as long as the INSERT statement references the columns by name, you can, using the INSERTED and/or DELETED virtual table in the INSTEAD OF trigger.

For example, create the following TABLE and VIEW objects as seen in Listing 2-13.

LISTING 2-13 Setting up a scenario for demonstrating using TRIGGER to make any view editable

Click here to view code image

```
CREATE TABLE Examples.KeyTable1
(
    KeyValue int NOT NULL CONSTRAINT PKKeyTable1 PRIMARY KEY,
    Value1   varchar(10) NULL
);
CREATE TABLE Examples.KeyTable2
(
    KeyValue int NOT NULL CONSTRAINT PKKeyTable2 PRIMARY KEY,
    Value2    varchar(10) NULL
);
GO
CREATE VIEW Examples.KeyTable
AS
```

```
    SELECT COALESCE(KeyTable1.KeyValue, KeyTable2.KeyValue) as
KeyValue,
           KeyTable1.Value1, KeyTable2.Value2
    FROM   Examples.KeyTable1
             FULL OUTER JOIN Examples.KeyTable2
                ON KeyTable1.KeyValue = KeyTable2.KeyValue;
```

Note that in the view, there is no way that you can insert data using this view, because the actual KeyValue columns are not exposed in the view, so the following attempt is to insert into the table:

Click here to view code image

```
INSERT INTO Examples.KeyTable (KeyValue, Value1, Value2)
VALUES (1,'Value1','Value2');
```

This gives you the following error:

Click here to view code image

```
Msg 4406, Level 16, State 1, Line 21
Update or insert of view or function 'Examples.KeyTable' failed
because it contains
a derived or constant field.
```

Next, add an INSTEAD OF INSERT TRIGGER to the table. In the TRIGGER, we get the inserted and deleted virtual tables that are the shape of the VIEW objects structure, which we will use to do the INSERT operations as seen in Listing 2-14.

LISTING 2-14 INSTEAD OF TRIGGER to make view editable

Click here to view code image

```
CREATE TRIGGER Examples.KeyTable_InsteadOfInsertTrigger
ON Examples.KeyTable
INSTEAD OF INSERT
AS
BEGIN
    SET NOCOUNT ON;
    SET ROWCOUNT 0; --in case the client has modified the
rowcount
    BEGIN TRY
          --Insert data into one of the tables
        INSERT INTO Examples.KeyTable1(KeyValue, Value1)
        SELECT KeyValue, Value1
        FROM   Inserted;
        --and then the other
        INSERT INTO Examples.KeyTable2(KeyValue, Value2)
```

```
        SELECT KeyValue, Value2
        FROM   Inserted;
    END TRY
    BEGIN CATCH
        IF XACT_STATE() <> 0
            ROLLBACK TRANSACTION;
        THROW; --will halt the batch or be caught by the
caller's catch block
    END CATCH;
END;
```

Now, if you try to insert into the view, using the same statement as before:

Click here to view code image

```
INSERT INTO Examples.KeyTable (KeyValue, Value1, Value2)
VALUES (1,'Value1','Value2');
```

It will succeed. And to view the data:

```
SELECT *
FROM   Examples.KeyTable;
```

It looks just like any other table:

Click here to view code image

```
KeyValue    Value1     Value2
----------- ---------- ----------
1           Value1     Value2
```

This is a very simple version of what this TRIGGER may need to be for a production worthy version. When you have more than one table to be concerned with, there is a question of what happens if one row already exists, and another doesn't. If you build the UPDATE trigger, an UPDATE can either be an UPDATE or an INSERT for one of the tables. But you should make sure that one of the table's rows exists.

Determine when to use Data Manipulation Language (DML) triggers, Data Definition Language (DDL) triggers, or logon triggers

As has been stated a few times already in this trigger skill (and in the name of this section itself) there are three types of triggers that activate upon the occurrence of a type of event. You have already seen that DML TRIGGER objects are schema bound database objects that let you react to, and even morph the results of, an INSERT, UPDATE or DELETE statement.

In this section, we review the other two types of TRIGGER objects:

- **DDL triggers** Used to react to DDL operations at the server or database level. For

example, you can capture the DDL of every CREATE TABLE and ALTER TABLE statement and log the results in a table, or even stop them from occurring.

- **Logon triggers** Used to react to someone logging into the server. For example, you could state that login LOGIN1 (also referred to as a *server principal*) could not log in from 8PM – 7AM. These triggers just stop the login action, so if the user is already connected during this time period, it does not end their connection.

> **Need More Review? DLL triggers**
>
> This section contains a high-level overview and a few example triggers for review. If you want the complete breadth of coverage on DDL triggers, here are some recommended resources. The following article by Aaron Bertrand: SQL Server DDL Triggers to Track All Database Changes: *https://www.mssqltips.com/sqlservertip/2085/sql-server-ddl-triggers-to-track-all-database-changes/* and this Technet article on DDL Triggers: *https://technet.microsoft.com/en-us/library/ms190989.aspx*.

DDL Triggers

There are two kinds of DDL Triggers that you can use to react to DDL statements. One is at the database level, where you can react to DDL only in the database where the trigger is located. For example, the trigger can fire on CREATE TABLE, DROP INDEX, ALTER VIEW, etc. The other is at the server level. Server triggers can react to actions that occur in any database, as well as things that occur strictly at the server scope, such as CREATE DATABASE, DROP LOGIN, etc.

In the next two sections, we cover examples of DDL triggers at the server, and then the database level. There is not that much difference to the syntax, but there are a few differences to understand about how they work, and where they reside.

Server

For the server scoped example, a TRIGGER object logs whenever a database is created, dropped, or altered. In this example, the location of the database of the log table is important, because a SERVER DDL TRIGGER object is stored at the server level in the master database. So the trigger needs to address the table by three-part name.

Create the following TABLE object that contains the time, statement, and the login name of the user that made the change to the database.

Click here to view code image

```
USE ExamBook762Ch2;
GO
CREATE TABLE Examples.DDLDatabaseChangeLog
(
```

```
    DDLDatabaseChangeLogId int NOT NULL IDENTITY
         CONSTRAINT PKDDLDatabaseChangeLog PRIMARY KEY,
     LogTime datetime2(0) NOT NULL,
     DDLStatement nvarchar(max) NOT NULL,
     LoginName sysname NOT NULL
);
```

Note Using the code

If you are trying out the code in this chapter yourself, be aware that making a mistake in the configuration of a DDL trigger can cause typical operations to fail.

Next, since the log table is in a different database, but the trigger is not scoped to that database, we need to create security principals. Security is not on the exam, but to make the example somewhat realistic, this is needed because security chaining does not pertain to this non-schema owned object. Instead use the EXECUTE AS clause on the CREATE TRIGGER statement to dictate security principals. We start by creating a server principal, a user in the ExamBook762Ch2 database.

Click here to view code image

```
--Names used to make it clear where you have used examples from
this book outside
--of primary database
CREATE LOGIN Exam762Examples_DDLTriggerLogging WITH PASSWORD =
'PASSWORD$1';
CREATE USER Exam762Examples_DDLTriggerLogging
                                    FOR LOGIN
Exam762Examples_DDLTriggerLogging;
GRANT INSERT ON  Examples.DDLDatabaseChangeLog TO
                               Exam762Examples_DDLTriggerLogging;
```

We use just three events to cover the database events listed in our requirements, but there are many more. For a full list, check this link to DDL Events on Technet: *https://technet.microsoft.com/en-us/library/bb522542.aspx*.

The trigger itself is fairly simple. The interesting part is the EVENTDATA() function. It returns an XML string value that contains information about the DDL operation that caused the trigger to fire. Unlike a DML TRIGGER object, a DDL TRIGGER fires once per statement. We use just a single value from the statement, the CommandText value. Note that the DDL TRIGGER object is not a schema-owned object.

Click here to view code image

```
CREATE TRIGGER DatabaseCreations_ServerDDLTrigger
ON ALL SERVER
```

```
WITH EXECUTE AS 'Exam762Examples_DDLTriggerLogging'
FOR CREATE_DATABASE, ALTER_DATABASE, DROP_DATABASE
AS
    SET NOCOUNT ON;
    --trigger is stored in master db, so must
    INSERT INTO
ExamBook762Ch2.Examples.DDLDatabaseChangeLog(LogTime,
DDLStatement,

    SELECT SYSDATETIME(),EVENTDATA().value(
                    '(/EVENT_INSTANCE/TSQLCommand/CommandText)
[1]','nvarchar(max)'),
           ORIGINAL_LOGIN(); --Original login gives you the
user that is connected.
                             --Otherwise we would get the
EXECUTE AS user.
```

To test this trigger, create a LOGIN and give it rights to create and alter a database, assuming your server allows SQL Standard logins. If not, the same information is captured if you use any login.

Click here to view code image

```
CREATE LOGIN Exam762Examples_DatabaseCreator WITH PASSWORD =
'PASSWORD$1';
GRANT CREATE ANY DATABASE TO Exam762Examples_DatabaseCreator;
GRANT ALTER ANY DATABASE TO Exam762Examples_DatabaseCreator;
```

Now, login as Exam762Examples_DatabaseCreator, and run the following set of batches:

Click here to view code image

```
CREATE DATABASE Example
GO
ALTER DATABASE Example SET RECOVERY SIMPLE;
GO
DROP DATABASE Example;
```

And the database owner that you have been using:

Click here to view code image

```
SELECT LogTime, DDLStatement, LoginName
FROM Examples.DDLDatabaseChangeLog;
```

You receive a log of changes:

Click here to view code image

```
LogTime                     DDLStatement
```

```
--------------------------- ----------------------------------
-------- ------------
2016-09-21 16:55:09         CREATE DATABASE
Example                      Exam762Examp...
2016-09-21 16:55:19         ALTER DATABASE Example SET RECOVERY
SIMPLE   Exam762Examp...
2016-09-21 16:55:27         DROP DATABASE
Example                        Exam762Examp...
```

While mostly an administration function, this provides functionality that can be very useful. Something you can do in a DDL TRIGGER is ROLLBACK to disallow an action (no need for special security here):

Click here to view code image

```
CREATE TRIGGER DatabaseCreations_StopThemAll
ON ALL SERVER
FOR CREATE_DATABASE, ALTER_DATABASE, DROP_DATABASE
AS
    SET NOCOUNT ON;
    ROLLBACK TRANSACTION;
    THROW 50000,'No more databases created please',1;
```

Now, everyone (even system administrators), is disallowed to change a database. Something is commonly done with DDL Triggers of this type is to disable them. You can disable a TRIGGER using the DISABLE TRIGGER statement:

Click here to view code image

```
DISABLE TRIGGER DatabaseCreations_StopThemAll ON ALL SERVER;
```

It is a very good idea to clean up your objects unless you want to keep them, as they span outside of the single database:

Click here to view code image

```
DROP TRIGGER DatabaseCreations_ServerDDLTrigger ON ALL SERVER;
DROP USER Exam762Examples_DDLTriggerLogging;
DROP LOGIN Exam762Examples_DDLTriggerLogging;
DROP LOGIN Exam762Examples_DatabaseCreator;
```

Database

There is very little difference between the DDL TRIGGER objects at the database scope versus the server scope. Pretty much the exact same syntax works, but there are fewer events to react to. In this example, we demonstrate another use for DDL TRIGGERS, and that is stopping an event, while logging it.

We start with a table that is very much the same as the one for database changes:

Click here to view code image

```
CREATE TABLE Examples.DDLChangeLog
(
   DDLChangeLogId int NOT NULL IDENTITY
        CONSTRAINT PKDDLChangeLog PRIMARY KEY,
    LogTime datetime2(0) NOT NULL,
    DDLStatement nvarchar(max) NOT NULL,
    LoginName sysname NOT NULL
);
```

We again need to configure some security so the user can perform an INSERT statement into the table, but this time strictly in the context of the database:

Click here to view code image

```
CREATE USER Exam762Examples_DDLTriggerLogging WITHOUT LOGIN;
GRANT INSERT ON Examples.DDLChangeLog TO
Exam762Examples_DDLTriggerLogging;
```

Now we create the DDL TRIGGER on the database scope. Just like the server version, this is not a schema-scoped object. In this version of the trigger we are going to save off the DDL into a variable, do the ROLLBACK TRANSACTION, and then log the change (note that if the DDL statement is in an external transaction, the change is still logged because of the ROLLBACK TRANSACTION).

Click here to view code image

```
CREATE TRIGGER DatabaseChanges_DDLTrigger
ON DATABASE
WITH EXECUTE AS 'Exam762Examples_DDLTriggerLogging'
FOR CREATE_TABLE, ALTER_TABLE, DROP_TABLE
AS
    SET NOCOUNT ON;
    DECLARE @eventdata XML = EVENTDATA();
    ROLLBACK; --Make sure the event doesn't occur
    INSERT INTO Examples.DDLChangeLog(LogTime, DDLStatement,
LoginName)
    SELECT SYSDATETIME(),
           @EventData.value('(/EVENT_INSTANCE/TSQLCommand/Comma
[1]',

           ORIGINAL_LOGIN();
    THROW 50000,'Denied!',1;
```

Now, when any user in the database (even a system administrator) tries to CREATE, ALTER, or DROP a TABLE object:

Click here to view code image

```
CREATE TABLE Examples.Test
(
```

```
    TestId int NOT NULL
);
GO
DROP TABLE Examples.DDLChangeLog;
```

You receive the following error message (or in this case, you would get two of the same error message):

Click here to view code image

```
Msg 50000, Level 16, State 1, Procedure
DatabaseChanges_DDLTrigger, Line 25
Denied!
```

Then, viewing the data in the log table:

Click here to view code image

```
SELECT LogTime, DDLStatement, LoginName
FROM   Examples.DDLChangeLog;
```

You see the following statements were attempted:

Click here to view code image

```
LogTime                     DDLStatement
--------------------------- ----------------------------------
- ----------------------
2016-09-21 19:16:06         CREATE TABLE
Examples.Test           DomainName\louis
                            (
                                 TestId int NOT NULL
                            )
2016-09-21 19:16:51         DROP TABLE
Examples.DDLChangeLog;    DomainName\louis
```

Again, clean up your code or future examples do not work:

Click here to view code image

```
DROP TRIGGER DatabaseChanges_DDLTrigger ON DATABASE;
DROP USER Exam762Examples_DDLTriggerLogging;
```

Logon Triggers

The last type of TRIGGER objects to introduce are LOGIN TRIGGER modules. A LOGIN TRIGGER fires whenever a server principal connects to your server. In our example, it implements the following requirements: disallow a server principal named: Login_NotAllowed from connecting to the server, and log all other connections in a log table.

For this example, create the following table, which can capture the name of the login, the

time of login, and the application that the login comes from:

Click here to view code image

```
CREATE TABLE Examples.LoginLog
(
    LoginLogId  int NOT NULL IDENTITY(1,1)
          CONSTRAINT PKLoginLog PRIMARY KEY,
    LoginName   sysname NOT NULL,
    LoginTime   datetime2(0) NOT NULL ,
    ApplicationName sysname NOT NULL
);
```

Similar to the DDL trigger, there is very little to the LOGIN TRIGGER. The trigger fires once per logon operation, and there is no data it provides by a function or virtual table. All data you need comes from system functions. In order to log on to a table, just like the DDL triggers, we need to provide security information, as this is a non-schema bound object with no database context.

Click here to view code image

```
CREATE LOGIN Exam762Examples_LogonTriggerLogging WITH PASSWORD
= 'PASSWORD$1';
CREATE USER Exam762Examples_LogonTriggerLogging
                                    FOR LOGIN
Exam762Examples_LogonTriggerLogging;
GRANT INSERT ON Examples.LoginLog TO
Exam762Examples_LogonTriggerLogging;
```

Now create the trigger. It uses the ORIGINAL_LOGIN() function to get the security context from the principal that connected, as the EXECUTE AS clause changes the context inside the trigger. Then, if the user is not Login_NotAllowed, it logs the data:

Click here to view code image

```
CREATE TRIGGER Exam762ExampleLogonTrigger
ON ALL SERVER
WITH EXECUTE AS 'Exam762Examples_LogonTriggerLogging'
FOR LOGON
AS
    IF ORIGINAL_LOGIN() = 'Login_NotAllowed'
        THROW 50000,'Unauthorized Access',1;
    ELSE
        INSERT INTO ExamBook762Ch2.Examples.LoginLog(LoginName,
LoginTime,
                                                     Applicatio
        VALUES (ORIGINAL_LOGIN(),SYSDATETIME(),APP_NAME());
```

Note LOGON TRIGGER errors

If you have errors in your LOGON TRIGGER (such as inserting into a table that the security context of the trigger creator cannot access), you can lock out every user, including members of the sysadmin role. You can bypass the LOGON TRIGGER by starting SQL Server in a minimal configuration (a startup parameter of -f, as described in the MSDN article: Database Engine Service Startup Options: *https://msdn.microsoft.com/en-us/library/ms190737.aspx*).

To test the LOGIN TRIGGER, create the LOGIN:

Click here to view code image

```
CREATE LOGIN Login_NotAllowed WITH PASS

WORD = 'PASSWORD$1';
```

Try to log in (see Figure 2-2), using any tool, such as Management Studio.

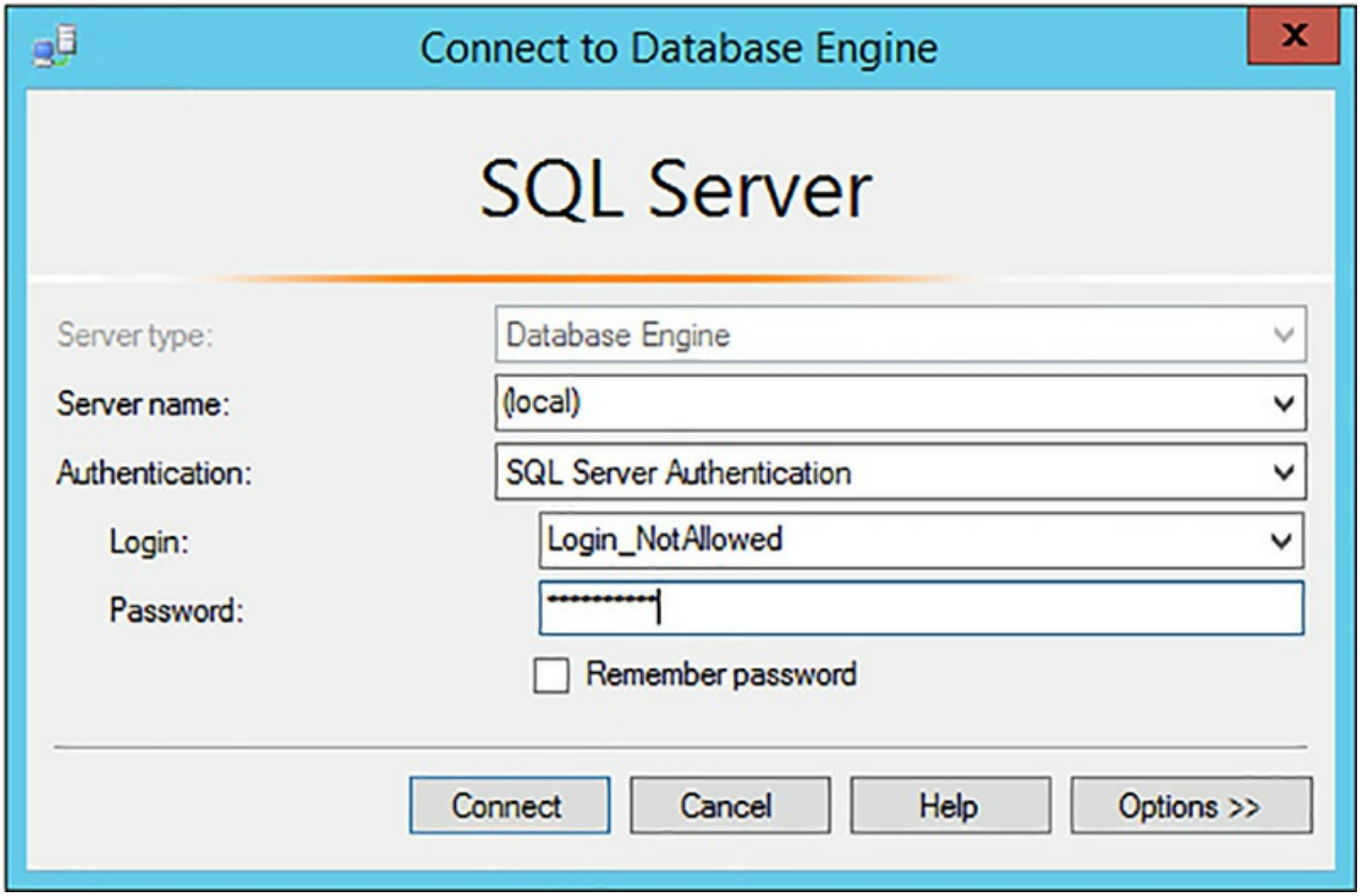


FIGURE 2-2 Connection dialog for SQL Server Management Studio

After clicking connect, you see the dialog in Figure 2-3.

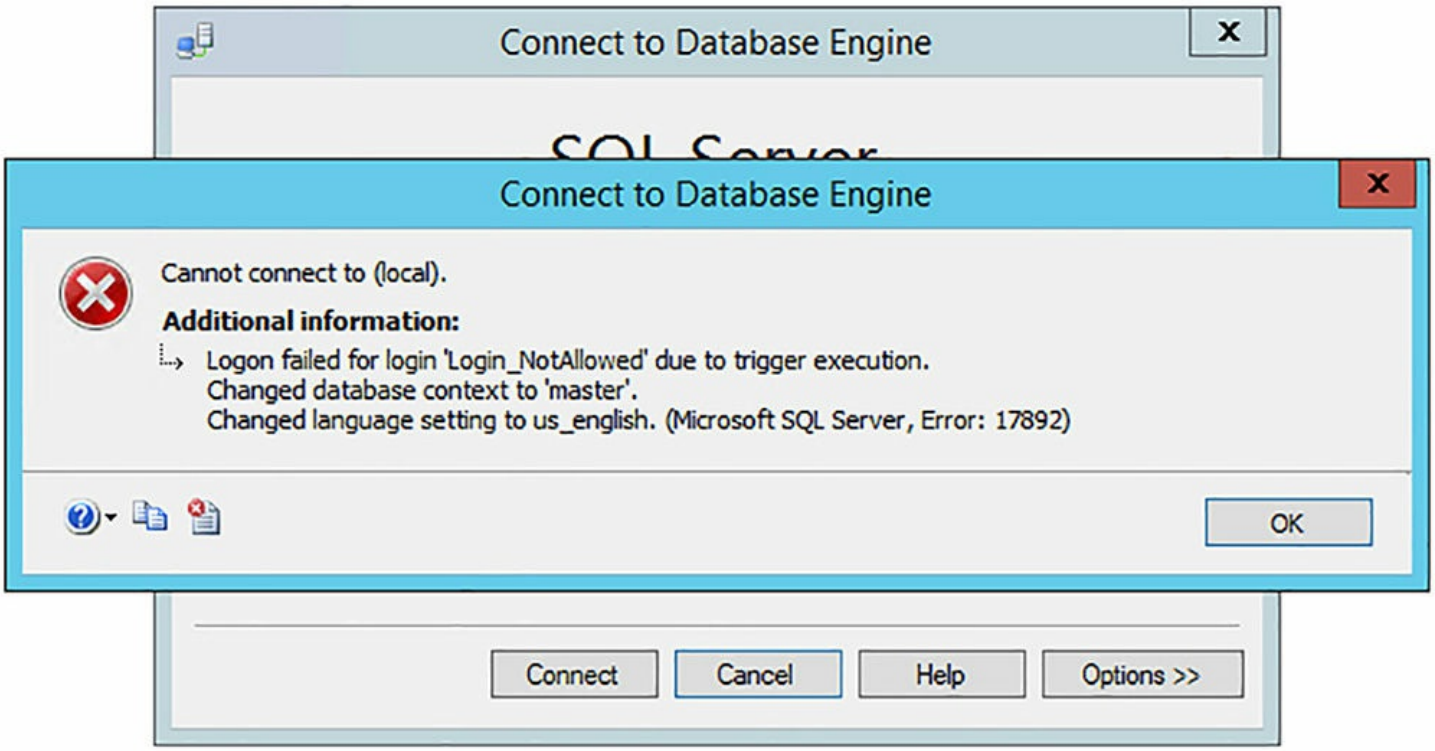


FIGURE 2-3 Failed connection dialog from SQL Server Management Studio when LOGON TRIGGER has prevented connection

To be sure that your LOGON TRIGGER actually works before disconnecting all of your connections, connect to a new window using your typical security rights. Don't be terribly surprised when you find this log has more rows than you initially expect:

Click here to view code image

```
   LoginName                 LoginTime           ApplicationNa
------------------------ -------------------- ----------------
----------------------
WIN-8F59BO5AP7D\louis    2016-09-21 21:26:50  Microsoft SQL
Se..IntelliSense
WIN-8F59BO5AP7D\louis    2016-09-21 21:26:50  Microsoft SQL
Se..IntelliSense
WIN-8F59BO5AP7D\louis    2016-09-21 21:27:06  Microsoft SQL
Se..Query
WIN-8F59BO5AP7D\louis    2016-09-21 21:27:06  Microsoft SQL
Se..Query
```

As with the previous examples, be sure to clean this up if you are doing this on your server, because if the ExamBook762Ch2 database is later dropped with this trigger enabled, no one will be able to logon to the server:

Click here to view code image

```
DROP TRIGGER Exam762ExampleLogonTrigger ON ALL SERVER;
DROP USER Exam762Examples_LogonTriggerLogging;
DROP LOGIN Exam762Examples_LogonTriggerLogging;
```

Need More Review? LOGON TRIGGERS

For more details on Logon Triggers, the following MSDN article covers this: *https://msdn.microsoft.com/en-us/library/bb326598.aspx*.

Recognize results based on execution of AFTER or INSTEAD OF triggers

It is important to be able to trace through code and understand how it works and what it does based on particular inputs. Here in the trigger section of the book it is no different, except that triggers are quite a bit more complex than any code we have tried before. In the following two examples, we present you with a couple of less realistic TABLE and DML TRIGGER object combinations, with a DML statement that performs some action. Your task is to determine what occurs as data passes through the TRIGGER object (before reading on to see the solution directly after the example, naturally).

The first example, uses an AFTER TRIGGER object. Start with the following table:

Click here to view code image

```
CREATE TABLE Examples.UpdateRows
(
    UpdateRowsId int NOT NULL IDENTITY(1,1)
        CONSTRAINT PKUpdateRows PRIMARY KEY,
    Value varchar(20) NOT NULL
);
INSERT INTO Examples.UpdateRows (Value)
VALUES ('Original'),('Original'),('Original');
```

This has the following data:

Click here to view code image

```
UpdateRowsId Value
------------ --------------------
1            Original
2            Original
3            Original
```

And the AFTER UPDATE TRIGGER object in Listing 2-15 is added to the table.

LISTING 2-15 Trigger for the AFTER TRIGGER example

Click here to view code image

```
CREATE TRIGGER Examples.UpdateRows_TriggerInsert
ON Examples.UpdateRows
AFTER UPDATE AS
BEGIN
```

```
   SET NOCOUNT ON;
   SET ROWCOUNT 0;
   BEGIN TRY
        DECLARE @UpdateRowsId int
        SELECT @UpdateRowsId = UpdateRowsId
        FROM   inserted
        ORDER BY UpdateRowsId;

        UPDATE Examples.UpdateRows
        SET    Value = UPPER(Value)
        WHERE  UpdateRowsId = @UpdateRowsId;
   END TRY
   BEGIN CATCH
       IF XACT_STATE() <> 0
           ROLLBACK TRANSACTION;
       THROW; --will halt the batch or be caught by the
caller's catch block
   END CATCH;

END;
```

Now, a user runs the following UPDATE statement:

Click here to view code image

```
UPDATE Examples.UpdateRows
SET    Value = 'Modified';
```

And receives the following output:

```
(3 row(s) affected)
```

What are the contents of the table? Either:

Click here to view code image

```
UpdateRowsId Value
------------ --------------------
1            Modified
2            Modified
3            MODIFIED

UpdateRowsId Value
------------ --------------------
1            Original
2            Original
3            Original
UpdateRowsId Value
------------ --------------------
```

```
1            MODIFIED
2            MODIFIED
3            MODIFIED

UpdateRowsId Value
------------ --------------------
1            MODIFIED
2            Modified
3            Modified
```

Or can you actually tell?

In this case, the fourth set of outputs match the table contents. Because of the way the TRIGGER is coded, only a single row is modified. So, it would be the first or the last set of results, or how you can't tell. You are guaranteed to get the first row in the set because of the ORDER BY clause on this statement (and it is necessary for you to realize that ORDER BY would order in ascending order by default):

Click here to view code image

```
SELECT @UpdateRowsId = UpdateRowsId
FROM   inserted
ORDER BY UpdateRowsId;
```

Without the ORDER BY clause, the order is not guaranteed, so you might get a different result (even though the fourth result would still be extremely likely).

This second example uses an INSTEAD OF TRIGGER object. Previous examples of triggers have made sure that the primary key column value was not changeable by using a column with the IDENTITY property or that it did not matter. The table we use is very simple, and the PRIMARY KEY constraint is on a column that can be changed:

Click here to view code image

```
CREATE TABLE Examples.KeyModify
(
    KeyModifyId  int CONSTRAINT PKKeyModify PRIMARY KEY,
    Value        varchar(20)
);
INSERT INTO Examples.KeyModify(KeyModifyId, Value)
VALUES (1,'Original'), (2,'Original'),(3,'Original');
```

Now, the trigger in Listing 2-16 is added to the table.

LISTING 2-16 Trigger for the INSTEAD OF TRIGGER example

Click here to view code image

```
CREATE TRIGGER Examples.KeyModify_TriggerInsteadOfInsert
```

```
ON Examples.KeyModify
INSTEAD OF UPDATE AS
BEGIN
   SET NOCOUNT ON;
   SET ROWCOUNT 0;
   BEGIN TRY
        UPDATE Examples.KeyModify
        SET    Value = UPPER(inserted.Value)
        FROM   Examples.KeyModify
                 JOIN inserted
                    ON KeyModify.KeyModifyId =
inserted.KeyModifyId
   END TRY
   BEGIN CATCH
       IF XACT_STATE() <> 0
           ROLLBACK TRANSACTION;
       THROW;
   END CATCH
END;
```

Now, a user runs the following statement:

Click here to view code image

```
UPDATE Examples.KeyModify
SET     KeyModifyId = KeyModifyId + 10, --Change Primary Key
Value
        Value = 'Modified';
```

After performing this statement, the user gets the following message:

```
(3 row(s) affected)
```

Did anything change in the table? Are the KeyModifyId column values 11, 12, and 13? Does the Value column say Original, ORIGINAL, Modified, or MODIFIED? The key to the answer is the JOIN on the inserted virtual table. The inserted virtual table looks like:

Click here to view code image

```
KeyModifyId Value
----------- --------------------
11          Modified
12          Modified
13          Modified
```

This is just what your statement told it to do. However, the TABLE still has the following row values, since this is an INSTEAD OF TRIGGER object:

Click here to view code image

```
KeyModifyId Value
----------- --------------------
1           Original
2           Original
3           Original
```

At the point of the UPDATE. When you try to join these two sets together, zero rows match, so the final table looks like:

Click here to view code image

```
KeyModifyId Value
----------- --------------------
1           Original
2           Original
3           Original
```

The message from the UPDATE statement execution: (3 row(s) affected) is actually returned even if you leave off the UPDATE altogether. So nothing is changed. If it seems tricky, perhaps it is. However, they are real tables, and the code does work without error. You just have to work through the solution, regardless of whether it is realistic or not.

Design scalar-valued and table-valued user-defined functions based on business requirements

User-defined functions are schema-owned objects (in the same name space as tables, procedures, triggers, etc.) that can be used to encapsulate code in ways that can be used very naturally in Transact-SQL calls. There are two major kinds of user defined functions:

- **Scalar** Used to create code that returns a single value of a data type (more than just integers like STORED PROCEDURES could).
- **Table** To the user, it appears to be essentially a view that allows you to predefine parameters that can be used to filter or alter the output.

Each function has different uses, so let's review them independently.

> **Need More Review?**
>
> Beyond the review in this book, there is excellent information in the MSDN articles: User-Defined Functions (*https://msdn.microsoft.com/en-us/library/ms191007.aspx* and Create User-defined Functions (Database Engine) (*https://msdn.microsoft.com/en-us/library/ms191320.aspx*).

Scalar-Valued user-defined functions

Scalar UDFs allow you to encapsulate small amounts of code into a module that can be called inside of other statements. As a very simple example, the following scalar function

takes a single value as a parameter and returns it.

Click here to view code image

```
CREATE FUNCTION Examples.ReturnIntValue
(
    @Value  int
)
RETURNS int
AS
  BEGIN
    RETURN @Value
  END;
```

This can be called in a very similar manner to any system function we have used previously, for example:

Click here to view code image

```
SELECT Functions.ReturnIntValue(1) as IntValue;
```

You can access data in the function code, and they are very similar in structure to STORED PROCEDURE objects. The code in the FUNCTION has two primary limitations that make them complex to use.

First, there is no error handling that you can provide. There are runtime errors, such as those that stop a query from returning rows to occur that can't be caught during the compile process. One such example is the divide-by-zero errors. Perform the following query in the WideWorldImporters database:

Click here to view code image

```
SELECT OrderId, 1/ (4732-OrderId)
FROM   Sales.Orders;
```

You then see multiple rows returned and a divide-by-zero error. The same sort of issues occur with a scalar UDF, in that the errors come as data is being returned, not like you saw when creating STORED PROCEDURE objects. You cannot perform a THROW or RAISERROR statement to cause an error message to occur. Any error handling that you implement needs to be an understanding with the user of the function of the illegal value, like a negative, or NULL value.

Second, you may not make any side effects from the function. So you have no INSERT, UPDATE, or SELECT statements that modify tables other than a table variable (Of the form: DECLARE @table table(column datatype), which we use later in the table valued function code. They are available in scalar UDFs, but arern't typical, nor do any use of system functions change data.

Note that we did not use SET NOCOUNT ON, because that is considered a side-effecting function, even if it is simply for the scope of the object. BEGIN and END are

required around the body of the code, and you must have a RETURN statement that returns a value of the data type that matches the RETURNS clause.

It is allowable to access a table in your scalar functions. For example, still in the WideWorldImporters database, if you have a business requirement to implement a scalar UDF, the user can give them the number of orders for a Customer. Optionally, for a specific OrderDate, you can write the function shown in Listing 2-17.

LISTING 2-17 Scalar function that accesses a table

Click here to view code image

```
CREATE FUNCTION Sales.Customers_ReturnOrderCount
(
    @CustomerID int,
    @OrderDate date = NULL
)
RETURNS INT
WITH RETURNS NULL ON NULL INPUT, --if all parameters NULL,
return NULL immediately
     SCHEMABINDING --make certain that the tables/columns
referenced cannot change
AS
  BEGIN
      DECLARE @OutputValue int

      SELECT  @OutputValue = COUNT(*)
      FROM    Sales.Orders
      WHERE   CustomerID = @CustomerID
        AND   (OrderDate = @OrderDate
               OR @OrderDate IS NULL);

      RETURN @OutputValue
   END;
```

Using parameters of a FUNCTION object differs from using a STORED PROCEDURE, in that you can't use named parameters, and you can't skip parameters that have defaults. For example, to use this function you might code the following:

Click here to view code image

```
SELECT Sales.Customers_ReturnOrderCount(905, '2013-01-01');
```

This tells you that this customer has two orders for that day. To use the default parameter, you need to use the DEFAULT keyword:

Click here to view code image

```
SELECT Sales.Customers_ReturnOrderCount(905, DEFAULT);
```

While this can be quite useful, using functions in a query tends to cost more to perform over more direct manners. Consider the following two queries:

Click here to view code image

```
SELECT CustomerID, Sales.Customers_ReturnOrderCount(905,
DEFAULT)
FROM   Sales.Customers;

SELECT CustomerID, COUNT(*)
FROM   Sales.Orders
GROUP  BY CustomerID;
```

If you compare the plans, the first plan (which looks very complex graphically) is 3 percent of the cost of the two queries. The second query's plan is very simple-looking, but at 97 percent of the cost, it seems the hands down winner is the FUNCTION. But using SET STATISTICS ON you discover why.

Query using the FUNCTION:

Click here to view code image

```
Table 'Worktable'. Scan count 0, logical reads 0, physical
reads
Table 'Customers'. Scan count 1, logical reads 4, physical
reads
 SQL Server Execution Times:
    CPU time = 375 ms,  elapsed time = 439 ms.
```

Query using the SELECT with the GROUP BY:

Click here to view code image

```
Table 'Orders'. Scan count 1, logical reads 191, physical reads
0
SQL Server Execution Times:
   CPU time = 16 ms,  elapsed time = 34 ms.
```

The GROUP BY query looks far worse, but performs over 10 times faster. However, we know that the function is accessing the Orders table, and that information is missing. The same is true in the plan. The code in the scalar UDF is not represented in a straightforward manner in the plan either.

The most common use case for scalar UDFs is to format some data in a common manner. For example, say you have a business need to format a value, such as the CustomerPurchaseOrderNumber in the Sales.Orders table in WideWorldImporters in a given way, and in multiple locations. In this case we just right pad the data to eight characters, and prepend 'CPO' to the number. For this you can write an expression in the

SELECT clause:

Click here to view code image

```
SELECT N'CPO' + RIGHT(N'00000000' +
CustomerPurchaseOrderNumber,8)
FROM WideWorldImporters.Sales.Orders;
```

Now, if you need to use this in multiple places, you can fold that expression into a scalar USER DEFINED FUNCTION object, like so:

Click here to view code image

```
CREATE FUNCTION Sales.Orders_ReturnFormattedCPO
(
    @CustomerPurchaseOrderNumber nvarchar(20)
)
RETURNS nvarchar(20)
WITH RETURNS NULL ON NULL INPUT,
     SCHEMABINDING
AS
 BEGIN
    RETURN (N'CPO' + RIGHT(N'00000000' +
@CustomerPurchaseOrderNumber,8));
 END;
```

Now you can write:

Click here to view code image

```
SELECT Sales.Orders_ReturnFormattedCPO('12345') as
CustomerPurchaseOrderNumber;
```

This then returns:

```
CustomerPurchaseOrderNumber
---------------------------
CPO00012345
```

Note that this, too, has performance implications that are not quite as obvious as the function that accesses a table. First, you never want to use this to format a column in a WHERE clause:

Click here to view code image

```
SELECT OrderId
FROM   Sales.Orders
WHERE  Sales.Orders_ReturnFormattedCPO(CustomerPurchaseOrderNum
= 'CPO00019998';
```

In the best case, this scans an index that contains CustomerPurchaseOrder, but in the worst case it scans the entire base table structure. Note that this is true of any system

function as well, so it is not really just a general rule of thumb that any column values that are formatted in any clause other than the SELECT clause may be cause for concern with performance.

However, even in the SELECT clause, there is some overhead with using a scalar UDF:

Click here to view code image

```
SET STATISTICS TIME ON;
SELECT
Sales.Orders_ReturnFormattedCPO(CustomerPurchaseOrderNumber)
FROM   Sales.Orders;

SELECT N'CPO' + RIGHT(N'00000000' +
[CustomerPurchaseOrderNumber],8)
FROM WideWorldImporters.Sales.Orders;
```

In this test, the function version took 188 ms of CPU time, and the expression only 15ms. So whether or not it is worth it to use a scalar UDF is a personal preference. So, an exam question about scalar functions can be about what you include in the function, or it can ask you to predict the better-performing statement and/or why it might be the case.

Table-Valued user-defined functions

Table-Valued UDFs are used to present a set of data as a table, much like a view. In fact, they are generally thought of as views with parameters (or parameterized views.) There are two kinds of table-valued UDFs:

- **Simple** Consisting of a single Transact-SQL query, simple table-valued UDFs work very much like a VIEW.
- **Multi-Statement** Consists of as many statements as you need, allowing you to build a set of data using the same logic as you had in scalar UDFs, but returning a table instead of a scalar variable.

For these examples, use the same requirements used in our scalar example, returning the number of sales for a given customer, and optionally on a given day. In addition, add a requirement to determine if they have any backorders on that day.

Starting with the simple table-valued UDF, the basics of the object is, just like a VIEW, a single SELECT query. As such, there is not a performance penalty in using a table-valued USER DEFINED FUNCTION versus a VIEW, depending on how you use it (which can also be said about how VIEW objects are used.):

Click here to view code image

```
CREATE FUNCTION Sales.Customers_ReturnOrderCountSetSimple
(
    @CustomerID int,
    @OrderDate date = NULL
```

```
)
RETURNS TABLE
AS
RETURN (SELECT COUNT(*) AS SalesCount,
               CASE WHEN MAX(BackorderOrderId) IS NOT NULL
                         THEN 1 ElSE 0 END AS HasBackorderFlag
        FROM   Sales.Orders
        WHERE  CustomerID = @CustomerID
        AND   (OrderDate = @OrderDate
               OR @OrderDate IS NULL));
```

The syntax is pretty self-explanatory, you just declare that you are returning a table, and in the RETURN clause (no BEGIN and END), you put the query with the parameters used as you desire. Usage is much like a view, only you have parameters you need to include:

Click here to view code image

```
SELECT *
FROM   Sales.Customers_ReturnOrderCountSetSimple(905,'2013-01-
01');
```

This returns the following set:

Click here to view code image

```
SalesCount  HasBackorderFlag
----------- ----------------
2           1
```

And to default a parameter, you use the DEFAULT keyword as before:

Click here to view code image

```
SELECT *
FROM   Sales.Customers_ReturnOrderCountSetSimple(905,DEFAULT);
```

This returns:

Click here to view code image

```
SalesCount  HasBackorderFlag
----------- ----------------
125         1
```

Now you can use it in a query to get both calculated values by joining using the OUTER APPLY join operator, which applies column values from the left input as parameters into the right (you can also use literals):

Click here to view code image

```
SELECT CustomerId, FirstDaySales.SalesCount,
FirstDaySales.HasBackorderFlag
```

```
FROM   Sales.Customers
        OUTER APPLY Sales.Customers_ReturnOrderCountSetSimple
                              (CustomerId, AcountOpenedDate) as
FirstDaySales
WHERE  FirstDaySales.SalesCount > 0;
```

There are two APPLY operator versions. OUTER APPLY returns every row from the left input, while CROSS APPLY only returns rows where there is a match in the right input. Performing this query returns the following abridged output:

Click here to view code image

```
CustomerId  SalesCount  HasBackorderFlag
----------- ----------- ----------------
10          2           1
57          1           0
...         ...         ...
995         2           1
1000        2           1
```

For a multi-statement table-valued UDF, the syntax is quite different. You define the output specifically by declaring a table variable, and then by loading it. The following code in Listing 2-18 returns the exact same base query used in the simple version of the function.

LISTING 2-18 Multi-statement table-valued function that accesses a table

Click here to view code image

```
CREATE FUNCTION Sales.Customers_ReturnOrderCountSetMulti
(
    @CustomerID int,
    @OrderDate date = NULL
)
RETURNS  @OutputValue TABLE (SalesCount int NOT NULL,
                             HasBackorderFlag bit NOT NULL)
AS
 BEGIN
    INSERT INTO @OutputValue (SalesCount, HasBackorderFlag)
    SELECT COUNT(*) as SalesCount,
                  CASE WHEN MAX(BackorderOrderId) IS NOT NULL
                            THEN 1 ElSE 0 END AS
HasBackorderFlag
    FROM   Sales.Orders
    WHERE  CustomerID = @CustomerID
    AND   (OrderDate = @OrderDate
           OR @OrderDate IS NULL)
```

```
    RETURN;
END;
```

Multi-statement table-valued UDFs are always slower than equivalent simple ones. If you compare the plan and STATISTICS TIME output of the two queries, you see very similar issues with multi-statement table-valued UDFs, as there was with scalar UDFs that accessed tables. Compare the following two calls

Click here to view code image

```
SET STATISTICS TIME ON;
SELECT CustomerId, FirstDaySales.SalesCount,
FirstDaySales.HasBackorderFlag
FROM   Sales.Customers
        OUTER APPLY Sales.Customers_ReturnOrderCountSetSimple
                        (CustomerId, AccountOpenedDate) as
FirstDaySales
WHERE  FirstDaySales.SalesCount > 0;

SELECT CustomerId, FirstDaySales.SalesCount,
FirstDaySales.HasBackorderFlag
FROM   Sales.Customers
        OUTER APPLY Sales.Customers_ReturnOrderCountSetMulti
                        (CustomerId, AccountOpenedDate) as
FirstDaySales
WHERE  FirstDaySales.SalesCount > 0;
```

Note that the first plan that uses the simple form, is considered 89 percent of the cost. Yet when you look at the execution time, it takes twice as long. This is because the simple form is optimized like a VIEW object, incorporating the DDL of the object into the query plan, but the multi-statement form hides the costs of the coded object.

Identify differences between deterministic and non-deterministic functions

The term deterministic is a mathematics term that indicates that a system or equation that always returns the same value. This is important when building a FUNCTION, because the query optimizer can know that if one use of FUNCTION(1) returns 2, then the second performance of FUNCTION(1) returns 2.

In the system functions, some examples of deterministic functions are ABS, which returns the absolute value of a number, and YEAR, which returns the year from a date value. Functions that are not deterministic include SYSDATETIME(), which returns the current date and time, and NEWID(), which returns a new GUID value. For more information about deterministic and non-deterministic functions, the following MSDN article provides more details and functions *https://msdn.microsoft.com/en-us/library/ms178091.aspx*. The basic criteria is that the USER DEFINED FUNCTION is

declared as WITH SCHEMABINDING, accesses no external data, and uses no non-deterministic system functions.

One place this is important is when you are using a value in an index, either in a VIEW object or computed column in a TABLE object. So, when building functions, it is generally important to make your function deterministic. For example, consider the requirement you might have to build a function that proper cases a value by making the first letter in every word uppercase. Listing 2-19 includes a version of a function that does this.

LISTING 2-19 Slightly complex scalar function to demonstrate determinism

Click here to view code image

```
CREATE FUNCTION Examples.UpperCaseFirstLetter
(
    @Value varchar(50)
)
RETURNS nvarchar(50)
WITH SCHEMABINDING
AS
BEGIN
   --start at position 2, as 1 will always be uppercase if it
exists
   DECLARE @OutputValue nvarchar(50), @position int = 2,
@previousPosition int
   IF LEN(@Value) = 0 RETURN @OutputValue;
                       --remove leading spaces, uppercase the
first character
   SET @OutputValue =
(LTRIM(CONCAT(UPPER(SUBSTRING(@Value,1,1)),
                                            LOWER(SUBSTRING(@Val
   --if no space characters, exit
   IF CHARINDEX(' ',@OutputValue,1) = 0 RETURN @OutputValue;
   WHILE 1=1
    BEGIN
       SET @position = CHARINDEX(' ',@outputValue,@position) +
1
       IF @position < @previousPosition or @position = 0
           BREAK;
       SELECT @OutputValue =
CONCAT(SUBSTRING(@OutputValue,1,@position - 1),
                                    UPPER(SUBSTRING(@OutputValue
                                    SUBSTRING(@OutputValue,@posi
+ 1,50)),
             @PreviousPosition = @Position
    END
    RETURN @OutputValue
```

```
END;
```

You can run it as:

Click here to view code image

```
SELECT Examples.UpperCaseOnlyFirstLetter(N'NO MORE YELLING') as
Name;
```

This returns:

Click here to view code image

```
Name
-------------------------------------------------
No More Yelling
```

To determine if the FUNCTION is deterministic, use the OBJECTPROPERTY() function:

Click here to view code image

```
SELECT
OBJECTPROPERTY(OBJECT_ID('Examples.UpperCaseFirstLetter'),
'IsDeterministic')

IsDeterministic
```

No matter how complex it appears, since we did not use external data or non-deterministic system functions, and used WITH SCHEMABINDING, we discover it is deterministic:

```
IsDeterministic
---------------
1
```

For a non-deterministic example, consider the following function that gives you the start of the current month. It does this by using the SYSDATETIME() system function, which is non-deterministic:

Click here to view code image

```
CREATE FUNCTION Examples.StartOfCurrentMonth
()
RETURNS date
WITH SCHEMABINDING
AS
 BEGIN
    RETURN (DATEADD(day, 0, DATEDIFF(day, 0, SYSDATETIME() ) -
                                          DATEPART(DAY,SYSDATETIM
+ 1));
```

```
END;
```

And, we test if it is deterministic:

Click here to view code image

```
SELECT
OBJECTPROPERTY(OBJECT_ID('Examples.StartOfCurrentMonth'),
'IsDeterministic')

IsDeterministic
```

As expected, this function is not deterministic. While it might seem the case, determinism is not limited to scalar UDFs. Table-valued functions can be deterministic as well. Consider the following multi-statement table-valued UDF. It declares a table, loads it from a simple row constructor of 10 values, and returns the following:

Click here to view code image

```
CREATE FUNCTION Examples.ReturnOneToTenSet
()
RETURNS @OutputTable TABLE (I int)
WITH SCHEMABINDING
AS
  BEGIN
    INSERT INTO @OutputTable(I)
    VALUES (1),(2),(3),(4),(5),(6),(7),(8),(9),(10);

    RETURN;
  END;
```

Checking this function, you see that it is deterministic :

Click here to view code image

```
SELECT OBJECTPROPERTY(OBJECT_ID('Examples.ReturnOneToTenSet'),
'IsDeterministic')

IsDeterministic;
```

Chapter summary

- There are several types of constraints that you can use to help ensure data integrity in a database:
  - **PRIMARY KEY** Used to specify the primary uniqueness criteria for a table.
  - **UNIQUE** Used to enforce any additional uniqueness criteria other than the PRIMARY KEY constraint
  - **FOREIGN KEY** Enforces relationships between tables, making sure references exist. Usually references the PRIMARY KEY constraint, but can reference a

UNIQUE constraint as well.

- **CHECK** Allows you to declaratively specify Boolean predicates that must not be FALSE.
- **DEFAULT** Guides the user's input when there isn't necessary a simple choice for a value.

- **NULL** values are complicated with constraints. In UNIQUE constraints, they are treated as unique values. In CHECK constraints, they always pass the test unless explicitly tested for. For FOREIGN KEY constraints, they are always allowed, even if it is only one NULL column value in a composite key.
- There are two main ways to pick which columns to place a PRIMARY KEY constraint. Using a natural key, or a value from the logical group of attributes is one way. A very typical implementation is to use a surrogate key, usually some artificial value like an auto-generated value.
- STORED PROCEDURE objects are modules that allow you to create custom code that is performed together. A query plan is saved off with the stored procedure that is parameterized much easier than an ad-hoc batch of Transact-SQL.
- Using STORED PROCEDURE objects for building a coded interface to Transact-SQL objects allows programmers to do simple tasks in a manner similar to procedural programming languages.
- Use table-valued parameters to send a STORED PROCEDURE object many rows at a time, allowing you to create complex objects in single STORED PROCEDURE calls.
- For error handling, using the TRY...CATCH construct allows you to capture errors thrown by Transact-SQL statements. You can use THROW and RAISERROR to throw your own error messages. Unhandled THROW statements stop the batch from running, RAISERROR does not.
- TRIGGER objects can be used to react to different actions on the server. There are three kinds of triggers:
  - **DML** Used to enhance data integrity with access to more data than CHECK constraints, cascade modifications with more control than FOREIGN KEY constraints, and manipulate the data that is being inserted and updated into a table. There are INSTEAD OF triggers where you have to redo the action, and AFTER triggers that fire after the operation.
  - **DDL** Used to capture and react to server or database level DDL statements.
  - **Logon** Used to take some action when a server principal accesses the server.
- USER DEFINED FUNCTION objects allow you to build modules that are used in other Transact-SQL statement in the same way a table or a system function is. There are two kinds: table-valued and scalar.

Thought Experiment

In this thought experiment, demonstrate your skills and knowledge of the topics covered in this chapter. You can find answers in the next section. You have been assigned to implement a database for Trey Research, capturing details of the subjects they use in their research. The following list is a set of tasks that needs to be completed in the implementation. Determine what tool you can use from all of the ones we have discussed here in this chapter, and consider writing an example to make sure you understand the concepts.

- Users provide an email address when they sign up. However, sometimes the same person creates multiple accounts with the same email address, causing issues with the validity of a research results set.
- You have a stored procedure that needs to run three INSERT statements, the first two of which should all complete or all not complete. The fifth should run no matter what, succeeding/failing independently. How would you code this?
- In the RecipientType column in the Recipient table, there have been values entered like “Dunno” and “Whatever,” which are not valid types of recipients. How can you make sure that the column does not include values that it should not?
- You need to make sure that a column that contains an offer code is always five characters long and uppercase. What tool (or tools) will you use to make sure that the string value is all uppercase letters, and how?
- You have given users rights to add indexes to a certain database, but you want to make sure that no indexes are added from 8:00AM to 10:00AM.
- You are building a complex stored procedure that can take 10-20 seconds for each execution, and much longer if the @checkAll parameter has a NULL value, a value it should never have.

Though Experiment Answer

This section contains the solution to the thought experiment. Each answer explains why the answer choice is correct. Users provide an email address when they sign up. However, sometimes the same person creates multiple accounts with the same email address, causing issues with the validity of a research results set.

- For this need, you want to use a UNIQUE constraint on an EmailAddress column of the table where you define a participant for the survey questions. For example, the partial table was originally created:

Click here to view code image

```
CREATE TABLE Examples.Respondent
(

     RespondentId int NOT NULL CONSTRAINT PKRespondent
PRIMARY KEY,
```

```
     EmailAddress  nvarchar(500) NOT NULL
);
```

- Adding the following constraint prevents the issue with duplicated data:

Click here to view code image

```
ALTER TABLE Examples.Respondent
     ADD CONSTRAINT AKRespondent UNIQUE (EmailAddress);
```

- You have a stored procedure that needs to run three INSERT statements, the first two of which should all complete or all not complete. The third INSERT should run no matter what, succeeding/failing independently. How do you code this?

  Say you have the following simple table:

Click here to view code image

```
CREATE TABLE Examples.ThreeInsert
(
       ThreeInsertId int CONSTRAINT PKThreeInsert PRIMARY
KEY
);
```

- You can code something like the following. In code destined for production work, you likely want to code nested TRY...CATCH blocks, and save the error messages from the first two INSERT statements in variables to throw at the end.

Click here to view code image

```
CREATE PROCEDURE Examples.ThreeInsert_Create
            @SecondValue int = 2 --Pass in 1 to and no data
is inserted

AS
    SET NOCOUNT ON;
    BEGIN TRY
        BEGIN TRANSACTION;
        INSERT INTO Examples.ThreeInsert (ThreeInsertId)
        VALUES (1);
       INSERT INTO Examples.ThreeInsert (ThreeInsertId)
       VALUES (@SecondValue);
       COMMIT TRANSACTION;
    END TRY
    BEGIN CATCH
        IF XACT_STATE() <> 0
            ROLLBACK TRANSACTION;
        --No THROW will mean no reporting of message
    END CATCH;

    INSERT INTO Examples.ThreeInsert (ThreeInsertId)
```

```
    VALUES (3);
```

- In the RecipientType column in the Recipient table, there have been values entered like “Dunno” and “Whatever,” which are not valid types of recipients. How can you make sure that the column does not include values that it should not?
  - This was a bit of a tricky question, but the true answer to the question as written is that you can’t defend against an unknown entry, without a specific domain of legal values. It would be possible to create a CHECK constraint that has a predicate of (RecipientType NOT IN (‘Dunno’,’Whatever’)) and you have stopped two values, but not all of them.
  - After defining the legal values, say: ‘Regular,’ ‘Special Handling;’ you can then handle this in one of two ways. Consider the following partial table:

Click here to view code image

```
CREATE TABLE Examples.Recipient
(
    RecipientType varchar(30) NOT NULL
);
```

- You can add a CHECK constraint such as:

Click here to view code image

```
ALTER TABLE Examples.Recipient
   ADD CONSTRAINT CHKRecipient_RecipientType
         CHECK (RecipientType IN ('Regular','Special
Handling'));
```

- An alternate solution is to use a domain TABLE object with a FOREIGN KEY constraint, such as:

Click here to view code image

```
CREATE TABLE Examples.RecipientType
(
       RecipientType varchar(30) NOT NULL CONSTRAINT
PKRecipientType PRIMARY KEY
);

INSERT INTO Examples.RecipientType(RecipientType)
VALUES ('Regular'),('Special Handling');

ALTER TABLE Examples.Recipient
      ADD CONSTRAINT FKRecipient_Ref_ExamplesRecipientType
      FOREIGN KEY (RecipientType) REFERENCES Examples.
RecipientType(RecipientType);
```

- You need to make sure that the offer code column is always uppercase, what tool (or

tools) would you use to make sure that the string value is all uppercase letters, and how?

- For this question, there are two answers that would equally achieve the goal of making sure the string value is all uppercase letters. For example, consider the following table:

Click here to view code image

```
CREATE TABLE Examples.Offer
(
    OfferCode char(5) NOT NULL
);
```

- Consider using an INSTEAD OF TRIGGER object. When you are doing the INSERT and UPDATE operations, you force the value to be uppercase; after making sure all of the characters are letters. The following is the INSERT trigger.

Click here to view code image

```
CREATE TRIGGER Examples.Offer_TriggerInsteadOfInsert
ON Examples.Offer
INSTEAD OF INSERT AS
BEGIN
   SET NOCOUNT ON;
   SET ROWCOUNT 0; --in case the client has modified the
rowcount
   BEGIN TRY
        IF EXISTS (SELECT *
                   FROM   inserted
                   WHERE  OfferCode NOT LIKE '[A-Z][A-Z][A-
Z][A-Z][A-Z]')
            THROW 50000,'An OfferCode is not all alpha
characters',1;

        --skip columns to automatically set
        INSERT INTO Examples.Offer (OfferCode)
        SELECT UPPER(OfferCode)
        FROM   inserted
   END TRY
   BEGIN CATCH
       IF XACT_STATE() <> 0
           ROLLBACK TRANSACTION;
       THROW; --will halt the batch or be caught by the
caller's catch block
   END CATCH
END;
```

- Another method is actually to use a CHECK constraint. The requirement is to make

sure the string is all uppercase coming from the user. You can do this by using an expression with a case sensitive or binary collation. To determine the collation of your database, you can check sys.databases:

Click here to view code image

```
SELECT collation_name
FROM sys.databases
WHERE  database_id = DB_ID();
```

- This returns:

```
collation_name
----------------------------
Latin1_General_100_CI_AS
```

- Change it to CS for the case sensitive version of Latin1_General_100 and use this in the CHECK constraint:

Click here to view code image

```
ALTER TABLE Examples.Offer
    ADD CONSTRAINT CHKOffer_OfferCode
       CHECK (OfferCode LIKE '[A-Z][A-Z][A-Z][A-Z][A-Z]'
                             COLLATE
Latin1_General_100_CS_AS);
```

- You have given user rights to add indexes to a certain database, but you want to make sure that no indexes are added from 8:00AM to 10:00AM.
  - Use a DDL Trigger, such as the following. Casting SYSDATETIME() as time gives us the time of day:

Click here to view code image

```
CREATE TRIGGER DatabaseChanges_DDLTrigger
ON DATABASE
WITH EXECUTE AS 'Exam762Examples_DDLTriggerLogging'
FOR CREATE_INDEX
AS
    SET NOCOUNT ON;
    IF CAST(SYSDATETIME() AS time) >= '08:00:00'
        AND CAST(SYSDATETIME() AS time) < '10:00:00'

    THROW 50000,'No indexes may be added between 8 and 10
AM',1;
```

- You are building a complex stored procedure that can take 10-20 seconds for each execution, and much longer if the @checkAll parameter has a NULL value, a value it should never have.

- For this, in the body of your STORED PROCEDURE, you would include a check for the parameter:

Click here to view code image

```
IF @checkAll IS NULL
      THROW 50000,'The value of @checkAll may not be
NULL',1;
```

Chapter 3. Manage database concurrency

In a typical environment, a database receives multiple requests to perform an operation and often these requests can occur concurrently. As an administrator, you must understand how SQL Server handles these requests by default and the available options for changing this default behavior. Your overarching goal is to prevent unexpected results, while enabling as many processes as possible.

The 70-762 exam tests your skills related to this goal of managing database concurrency. Here in Skill 3.1, we review the basic properties and behaviors of transactions in SQL Server and the role of transactions in high-concurrency databases. Skill 3.2 addresses the available options for managing concurrency in SQL Server by using isolation levels and explores in detail the differences between isolation levels as well as the effect each isolation level has on concurrent transactions, system resources, and overall performance. Then in Skill 3.3 we explore the tools at your disposal to better understand locking behavior in SQL Server and the steps you can take to remediate deadlocks. Skill 3.4 introduces memory-optimized tables as another option for improving concurrency by explaining the use cases for which this approach is best, how to optimize performance when tables are held in memory instead of on disk, and considerations for using and analyzing performance of natively compiled stored procedures.

Skills in this chapter:

- Implement transactions
- Manage isolation levels
- Optimize concurrency and locking behavior
- Implement memory-optimized tables and native stored procedures

Skill 3.1: Implement transactions

SQL Server protects data integrity by using transactions to control how, when, or even whether data changes in a database. **A** *transaction* is a unit of work consisting of one or more read and write commands that SQL Server executes completely or not at all. In the exam, you must be able to recognize scenarios in which transactions can complete successfully or not, and know how to use T-SQL statements to manage transaction behavior. You must also understand potential problems with transactions executing concurrently and how SQL Server uses locks to mitigate these problems.

This section covers how to:

- Identify DML statement results based on transaction behavior
- Recognize differences between and identify usage of explicit and implicit transactions

- Implement savepoints within transactions
- Determine the role of transactions in high-concurrency databases

Identify DML statement results based on transaction behavior

The results of a DML statement depends on transaction behavior. If the transaction succeeds, then the inserts, the updates, or the deletes that SQL Server executes as part of that transaction are committed to the database and permanently change the data in the affected tables. If the transaction fails for any reason, you can cancel or rollback the transaction to reverse any changes made to the database by the transaction prior to the failure. SQL Server has various methods for managing transaction behavior, but you also have options for changing this behavior when writing code to execute transactions.

In this section, we explore the ways that SQL Server supports the following set of properties collectively known in database theory as ACID to ensure data is protected in case of system or hardware failure:

- **Atomicity** An *atomic* transaction is a set of events that cannot be separated from one another and must be handled as a single unit of work. A common example is a bank transaction in which you transfer money from your checking account to your savings account. A successful atomic transaction not only correctly deducts the amount of the transfer from one account, but also adds it to the other account. If the transaction cannot complete all of its steps successfully, it must fail, and the database is unchanged.
- **Consistency** When a transaction is *consistent*, any changes that it makes to the data conform to the rules defined in the database by constraints, cascades, and triggers and thereby leave the database in a valid state. To continue the previous example, the amount removed from your checking account must be the same amount added to your savings account when the transaction is consistent.
- **Isolation** An isolated transaction behaves as if it were the only transaction interacting with the database for its duration. *Isolation* ensures that the effect on the database is the same whether two transactions run at the same time or one after the other. Similarly, your transfer to the savings account has the same net effect on your overall bank balances whether you were the only customer performing a banking transaction at that time, or there were many other customers withdrawing, depositing, or transferring funds simultaneously.
- **Durability** A *durable* transaction is one that permanently changes the database and persists even if the database is shut down unexpectedly. Therefore, if you receive a confirmation that your transfer is complete, your bank balances remain correct even if your bank experienced a power outage immediately after the transaction completed.

Note ACID property support

By default, SQL Server guarantees all four ACID properties, although you can request an alternate isolation level if necessary. We explain isolation levels in detail in Skill 3.2.

Before we start exploring transaction behavior, let's set up a new database, add some tables, and insert some data to establish a test environment as shown in Listing 3-1.

LISTING 3-1 Create a test environment for exploring transaction behavior

Click here to view code image

```
CREATE DATABASE ExamBook762Ch3;
GO
USE ExamBook762Ch3;
GO
CREATE SCHEMA Examples;
GO
CREATE TABLE Examples.TestParent
(
    ParentId  int NOT NULL
        CONSTRAINT PKTestParent PRIMARY KEY,
    ParentName  varchar(100) NULL
);

CREATE TABLE Examples.TestChild
(
    ChildId  int NOT NULL
        CONSTRAINT PKTestChild PRIMARY KEY,
    ParentId int NOT NULL,
    ChildName  varchar(100) NULL
);

ALTER TABLE Examples.TestChild
    ADD CONSTRAINT FKTestChild_Ref_TestParent
        FOREIGN KEY (ParentId) REFERENCES
Examples.TestParent(ParentId);

INSERT INTO Examples.TestParent(ParentId, ParentName)
VALUES (1, 'Dean'),(2, 'Michael'),(3, 'Robert');

INSERT INTO Examples.TestChild (ChildId, ParentId, ChildName)
VALUES (1,1, 'Daniel'), (2, 1, 'Alex'), (3, 2, 'Matthew'), (4,
3, 'Jason');
```

Even a single statement to change data in a table is a transaction (as is each individual INSERT statement in Listing 3-1). Consider this example:

```
UPDATE Examples.TestParent
SET ParentName = 'Bob'
WHERE ParentName = 'Robert';
```

When you execute this statement, if the system doesn't crash before SQL Server lets you know that the statement completed successfully, the new value is *committed*. That is, the change to the data resulting from the UPDATE statement is permanently stored in the database. You can confirm the successful change by running the following SELECT statement.

Click here to view code image

```
SELECT ParentId, ParentName
FROM Examples.TestParent;
```

The result of the UPDATE statement properly completed as you can see in the SELECT statement results.

```
ParentId   ParentName
---------- ------------
1          Dean
2          Michael
3          Bob
```

Atomicity

The execution of one statement at a time as a transaction does not clearly demonstrate the SQL Server support for the other ACID properties. Instead, you need a transaction with multiple statements. To do this, use the BEGIN TRANSACTION (or BEGIN TRAN) and COMMIT TRANSACTION (or COMMIT TRAN) statements (unless you implement implicit transactions as we describe in the next section).

You can test atomicity by attempting to update two different tables in the same transaction like this:

Click here to view code image

```
BEGIN TRANSACTION;
    UPDATE Examples.TestParent
    SET ParentName = 'Mike'
    WHERE ParentName = 'Michael';

    UPDATE Examples.TestChild
    SET ChildName = 'Matt'
    WHERE ChildName = 'Matthew';
COMMIT TRANSACTION;
```

When the transaction commits, the changes to both tables become permanent. Check the results with this query:

Click here to view code image

```
SELECT TestParent.ParentId, ParentName, ChildId, ChildName
FROM Examples.TestParent
    FULL OUTER JOIN Examples.TestChild ON TestParent.ParentId =
TestChild.ParentId;
```

The transaction updated both tables as you can see in the query results:

Click here to view code image

```
ParentId   ParentName   ChildId   ChildName
---------- ------------ --------- -----------
1          Dean         1         Daniel
1          Dean         2         Alex
2          Michael      3         Matt
3          Bob          4         Jason
```

On the other hand, if any one of the statements in a transaction fails, the behavior depends on the way in which you construct the transaction statements and whether you change the SQL Server default settings. A common misconception is that using BEGIN TRANSACTION and COMMIT TRANSACTION are sufficient for ensuring the atomicity of a transaction. You can test the SQL Server default behavior by adding or changing data in one statement and then trying to delete a row having a foreign key constraint in another statement like this:

Click here to view code image

```
        BEGIN TRANSACTION;
            INSERT INTO Examples.TestParent(ParentId,
  ParentName)
            VALUES (4, 'Linda');

        DELETE Examples.TestParent
        WHERE ParentName = 'Bob';
  COMMIT TRANSACTION;
```

In this case, the deletion fails, but the insertion succeeds as you can see by the messages that SQL Server returns.

Click here to view code image

```
(1 row(s) affected)
Msg 547, Level 16, State 0, Line 24
The DELETE statement conflicted with the REFERENCE constraint
"FKTestChild_Ref_
TestParent". The conflict occurred in database
```

```
"ExamBook762Ch3", table "Examples.
TestChild", column 'ParentId'.

The statement has been terminated.
```

When you check the data again, you see a total of four rows in the Examples.TestParent table:

```
ParentId    ParentName
----------- ------------
1           Dean
2           Michael
3           Bob
4           Linda
```

If you want SQL Server to roll back the entire transaction and thereby guarantee atomicity, one option is to use the SET XACT_ABORT option to ON prior to executing the transaction like this:

Click here to view code image

```
        SET XACT_ABORT ON;
        BEGIN TRANSACTION;
            INSERT INTO Examples.TestParent(ParentId, ParentName)
            VALUES (5, 'Isabelle');

    DELETE Examples.TestParent
    WHERE ParentName = 'Bob';
COMMIT TRANSACTION;
```

In this case, SQL Server rolls back all successfully completed statements in the transaction and returns the database to its state at the start of the transaction in which only four rows exist in the Examples.TestParent table as shown in the previous example. The SET XACT_ABORT option is set to OFF by default, therefore you must enable the option when you want to ensure that SQL Server rolls back a failed transaction.

What if the error raised is not a constraint violation, but a syntax error? Execute the following code that first disables the SET XACT_ABORT option (to prove the roll back works correctly with the default SQL Server setting) and then attempts an INSERT and a DELETE containing a deliberate syntax error:

Click here to view code image

```
        SET XACT_ABORT OFF;
        BEGIN TRANSACTION;
            INSERT INTO Examples.TestParent(ParentId, ParentName)
            VALUES (5, 'Isabelle');

    DELETE Examples.TestParent
```

```
    WHEN ParentName = 'Bob';
COMMIT TRANSACTION;
```

Although the INSERT is successful and would commit if the subsequent error were a constraint violation, SQL Server does not commit the insertion, and the database remains in its original state when it encounters a syntax error in a transaction.

Another option to consider is to explicitly include a roll back instruction in your transaction by enclosing it in a TRY block and adding a ROLLBACK TRANSACTION (or ROLLBACK TRAN) statement in a CATCH block:

Click here to view code image

```
BEGIN TRY
    BEGIN TRANSACTION;
        INSERT INTO Examples.TestParent(ParentId, ParentName)
        VALUES (5, 'Isabelle');

        DELETE Examples.TestParent
        WHERE ParentName = 'Bob';
    COMMIT TRANSACTION;
END TRY
BEGIN CATCH
    IF @@TRANCOUNT > 0 ROLLBACK TRANSACTION;
END CATCH
```

Because the transaction includes a DELETE statement that fails due to a constraint violation, the CATCH block is invoked and the transaction rolls back. Therefore, the Examples.Parent table still contains only four rows.

Notice also in the previous example that the execution of the ROLLBACK TRANSACTION requires the current status of the transaction (obtained by the @@TRANCOUNT variable) to be greater than 0, which means that a transaction is active. We explore the use of this variable in more detail in the section covering implicit and explicit transactions.

Exam Tip

For the exam, you should understand how nested transactions interact and how transactions roll back in the event of failure.

Need More Review? ROLLBACK TRANSACTION statement

For more in-depth information about the ROLLBACK TRANSACTION statement, see *https://msdn.microsoft.com/en-us/library/ms181299.aspx*.

Consistency

These last two examples not only demonstrate atomicity compliance in SQL Server, but also consistency. Another commonly used term for consistency is *data integrity*. To preserve data integrity in a database, you cannot remove a row from a table when there is an existing dependency on that row. Similarly, you cannot add a row to a table having foreign key constraints without providing a valid foreign key value in the new row. Any rule that you add to the database as we described in Chapter 2, "Implement programmability objects," is enforced by SQL Server to guarantee consistency.

Isolation

Now let's take a look at how SQL Server handles isolation by default. We explore your options for managing isolation in detail in Skill 3.2, but for Skill 3.1 you must understand what happens if you rely on the behavior of READ COMMITTED, the SQL Server default isolation level. To observe this behavior, set up two separate sessions in SQL Server Management Studio.

In one session, execute the following statement:

Click here to view code image

```
BEGIN TRANSACTION;
    INSERT INTO Examples.TestParent(ParentId, ParentName)
    VALUES (5, 'Isabelle');
```

The omission of the COMMIT statement in this example is deliberate. At this point, the transaction is still active, but it is not yet committed. Furthermore, the uncommitted transaction continues to hold a lock on the table preventing any other access to the table as long as the transaction remains uncommitted.

In the second session, execute the following statement:

```
SELECT ParentId, ParentName
FROM Examples.TestParent;
```

When you attempt to read rows from the locked table, the query continues to execute indefinitely because it is waiting for the transaction in the first session to complete. This behavior is an example of a write operation blocking a read operation. By default, SQL Server uses the READ COMMITTED isolation level to protect the transaction by preventing other operations from returning potentially incorrect results as a result of reading uncommitted inserts that could later be rolled back. It also insulates the transaction from premature changes to the values of those inserts by another transaction's update operation.

In the first session, end the transaction like this:

```
COMMIT TRANSACTION;
```

As soon as you commit the transaction, the query in the second session returns five rows and includes the newly inserted row:

```
ParentId   ParentName
---------- ------------
1          Dean
2          Michael
3          Bob
4          Linda
5          Isabelle
```

Durability

SQL Server guarantees full transaction durability by default. If the system crashes for some reason after SQL Server confirms a successful commit, the changes made by the transaction are visible after the system returns to an operable status even if the transaction operations had not been written to disk prior to the system failure.

To make this possible, SQL Server uses write-ahead logging to first hold data changes in a log buffer and then writes the changes to the transaction log on disk when the transaction commits or if the log buffer becomes full. The transaction log contains not only changes to data, but also page allocations and de-allocations, and changes to indexes. Each log record includes a unique log sequence number (LSN) to that every record change that belongs to the same transaction can be rolled back if necessary.

Once the transaction commits, the log buffer flushes the transaction log and writes the modifications first to the data cache, and then permanently to the database on disk. A change is never made to the database without confirming that it already exists in the transaction log. At that point, SQL Server reports a successful commit and the transaction cannot be rolled back.

What if a failure occurs after the change is written to the transaction log, but before SQL Server writes the change to the database? In this case, the data changes are uncommitted. Nonetheless, the transaction is still durable because you can recreate the change from the transaction log if necessary.

SQL Server also supports delayed durable transactions, also known as lazy commits. By using this approach, SQL Server can process more concurrent transactions with less contention for log IO, thereby increasing throughput. Once the transaction is written to the transaction log, SQL Server reports a successful transaction and any changes that it made are visible to other transactions. However, all transaction logs remain in the log buffer until the buffer is full or a buffer flush event occurs, at which point the transaction is written to disk and becomes durable. A buffer flush occurs when a fully durable transaction in the same database commits or a manual request to execute sp_flush_log is successful.

Delayed durability is useful when you are willing to trade potential data loss for reduced latency in transaction log writes and reduced contention between transactions.

Such a trade-off is acceptable in a data warehouse workload that runs batches frequently enough to pick up rows lost in a previous batch. The eventual resolution of data loss is acceptable alternative to durability only because the data warehouse is not the system of record. Delayed durability is rarely acceptable in an online transaction processing (OLTP) system.

Need More Review? Delayed transaction durability

You can enable a database to support delayed transaction durability and then force or disable delayed transaction durability at the transaction level as an option of the `COMMIT` statement. Although you should understand the concept and use cases for delayed durability for the exam, you do not need to identify all the possible options and interactions between database and transaction settings. However, if you would like more in-depth information about delayed transaction durability, refer to the MSDN description at *https://msdn.microsoft.com/en-us/library/ms181299.aspx*.

For an in-depth assessment of the performance and data loss implications of delayed transaction durability, see "Delayed Durability in SQL Server 2014" by Aaron Bertrand at *https://sqlperformance.com/2014/04/io-subsystem/delayed-durability-in-sql-server-2014*. Although the article was written for SQL Server 2014, the principles continue to apply to SQL Server 2016.

Recognize differences between and identify usage of explicit and implicit transactions

An important aspect of transaction management is knowing which commands are in scope. That is, you must know which commands are grouped together for execution as a single transaction. SQL Server supports the following methods for transaction control:

- **Auto-commit** Any single statement that changes data and executes by itself is automatically an atomic transaction. Whether the change affects one row or thousands of rows, it must complete successfully for each row to be committed. You cannot manually rollback an auto-commit transaction, although SQL Server performs a rollback if a system failure occurs before the transaction completes.
- **Implicit** An implicit transaction automatically starts when you execute certain DML statements and ends only when you use COMMIT TRANSACTION or ROLLBACK TRANSACTION. However, you must first configure a session to run in implicit transaction mode by first executing the SET IMPLICIT_TRANSACTIONS ON statement. After you do this, any of the following statements begin a new transaction: ALTER TABLE, BEGIN TRANSACTION, CREATE, DELETE, DROP, FETCH, GRANT, INSERT, OPEN, REVOKE, SELECT (only if selecting from a table),

TRUNCATE TABLE, and UPDATE.

- **Explicit** An explicit transaction has a specific structure that you define by using the BEGIN TRANSACTION at the beginning of the transaction and the COMMIT TRANSACTION or ROLLBACK TRANSACTION at the end of the transaction.

> **Need More Review? Batch-scoped transactions**
>
> QL Server also supports batch-scoped transactions when Multiple Active Result Sets (or MARS) is enabled, but you do not need to be familiar with this topic for the exam. If you would like to learn more about batch-scoped transactions, see *https://msdn.microsoft.com/en-us/library/ms131686.aspx*.

Implicit transactions

Let's examine the behavior of implicit transactions by executing a series of statements incrementally. First, enable the implicit transaction mode like this:

Click here to view code image

```
SET IMPLICIT_TRANSACTIONS ON;
```

Next, execute an INSERT statement and then check the status of open transactions:

Click here to view code image

```
INSERT INTO Examples.TestParent(ParentId, ParentName)
VALUES (6, 'Lukas');
SELECT @@TRANCOUNT;
```

The SELECT statement returns a 1 because SQL Server starts a new transaction when implicit transactions are enabled and the INSERT statement is executed. At this point, the transaction remains uncommitted and blocks any readers of the Examples.TestParent table.

Now you can end the transaction, check the status of open transactions, and check the change to the table by executing the following statements:

```
COMMIT TRANSACTION;
SELECT @@TRANCOUNT;
SELECT ParentId, ParentName
FROM Examples.TestParent;
```

The results of the SELECT statements show that the COMMIT statement both ended the transaction and decremented the @@TRANCOUNT variable and that a new row appears in the Examples.Parent table:

```
(No column name)
----------------
0
```

```
ParentId   ParentName
---------- ------------
1          Dean
2          Michael
3          Bob
4          Linda
5          Isabelle
6          Lukas
```

Important Transaction commitment by SQL Server

It is important to note that the transaction commits not only because the COMMIT statement is executed, but also because the value of @@TRANCOUNT is decremented to zero. Only at that time does SQL Server write log records and commit the transaction.

Now disable the implicit transaction mode:

Click here to view code image

```
SET IMPLICIT_TRANSACTIONS OFF;
```

Just as you can see in many of the transaction examples in the previous section, an implicit transaction can contain one or more statements and ends with an explicit execution of a COMMIT TRANSACTION or ROLLBACK TRANSACTION statement. Apart from the absence of a BEGIN TRANSACTION statement, an implicit transaction resembles an explicit transaction and behaves in the same way as well.

You might use implicit transactions when migrating an application from a different database platform or when you need to run your application across multiple database platforms because fewer code changes are required. In most cases, however, best practice dictates avoiding the use of implicit transactions. When you rely on auto-commit or explicit transactions instead, changes are committed as quickly as possible and performance is less likely to be adversely affected.

Exam Tip

For the exam, it is important to understand the impact of using implicit transactions. Be sure to review the remarks at "SET IMPLICIT_TRANSACTIONS (Transact-SQL)," *https://msdn.microsoft.com/en-us/library/ms187807.aspx*.

Explicit transactions

When you want complete control over transaction behavior, use an explicit transaction.

You have nothing to configure at the server or database level to enable explicit transactions. Simply enclose your transaction statements in the BEGIN TRANSACTION and COMMIT TRANSACTION statements. Furthermore, you should include logic to handle errors, such as a TRY/CATCH block, as shown in an example in the "Atomicity" section, or an IF/ELSE construct like this:

Click here to view code image

```
BEGIN TRANSACTION;
    INSERT INTO Examples.TestParent(ParentId, ParentName)
    VALUES (7, 'Mary');
    DELETE Examples.TestParent
    WHERE ParentName = 'Bob';
IF @@ERROR != 0
    BEGIN
        ROLLBACK TRANSACTION;
    RETURN
END
COMMIT TRANSACTION;
```

The following commands cannot be used in an explicit transaction:

- ALTER DATABASE
- ALTER FULLTEXT CATALOG
- ALTER FULLTEXT INDEX
- BACKUP
- CREATE DATABASE
- CREATE FULLTEXT CATALOG
- CREATE FULLTEXT INDEX
- DROP DATABASE
- DROP FULLTEXT CATALOG
- DROP FULLTEXT INDEX
- RECONFIGURE
- RESTORE

You can nest explicit transactions, although this capability is not ANSI-standard transaction behavior. As one example, consider a situation in which you have a set of statements in a transaction and one of the statements calls a stored procedure that starts its own transaction. Remember that each BEGIN TRANSACTION increments the @@TRANCOUNT variable and each COMMIT TRANSACTION decrements it. The ROLLBACK TRANSACTION resets the variable to zero and rolls back every statement to the beginning of the first transaction, but does not abort the stored procedure. When @@TRANCOUNT is zero, SQL Server writes to the transaction log. If the session ends

before @@TRANCOUNT returns to zero, SQL Server automatically rolls back the transaction.

Let's test this behavior by creating a stored procedure and calling it in a transaction as shown in Listing 3-2.

LISTING 3-2 Create and execute a stored procedure to test an explicit transaction

Click here to view code image

```
CREATE PROCEDURE Examples.DeleteParent
    @ParentId INT
AS
    BEGIN TRANSACTION;
        DELETE Examples.TestParent
        WHERE ParentId = @ParentId;
    IF @@ERROR != 0
        BEGIN
            ROLLBACK TRANSACTION;
            RETURN;
        END
    COMMIT TRANSACTION;
GO
BEGIN TRANSACTION;
    INSERT INTO Examples.TestParent(ParentId, ParentName)
    VALUES (7, 'Mary');
    EXEC Examples.DeleteParent @ParentId=3;
IF @@ERROR != 0
    BEGIN
        ROLLBACK TRANSACTION;
    RETURN
END
COMMIT TRANSACTION;
GO
```

When you execute these statements, several error messages display:

Click here to view code image

```
(1 row(s) affected)
Msg 547, Level 16, State 0, Procedure DeleteParent, Line 6
[Batch Start Line 16]
The DELETE statement conflicted with the REFERENCE constraint
"FKTestChild_Ref_TestParent". The conflict occurred in database
"ExamBook762Ch3", table
"Examples.TestChild", column 'ParentId'.
The statement has been terminated.
```

```
Msg 266, Level 16, State 2, Procedure DeleteParent, Line 0
[Batch Start Line 16]
Transaction count after EXECUTE indicates a mismatching number
of BEGIN and COMMIT
statements. Previous count = 1, current count = 0.
Msg 3903, Level 16, State 1, Line 25
The ROLLBACK TRANSACTION request has no corresponding BEGIN
TRANSACTION.
```

The first transaction begins with an INSERT statement at which point @@TRANCOUNT is 1. Then the call to the stored procedure results in the start of a second transaction and increments @@TRANCOUNT to 2. The constraint violation causes an error that then calls the ROLLBACK TRANSACTION statement, which in turn resets @@TRANCOUNT to 0 and rolls back the INSERT. The error message regarding the mismatching transaction count occurs because the @@TRANCOUNT value when the stored procedure ends no longer matches its value when the stored procedure started. That error leads to the ROLLBACK TRANSACTION statement in the first transaction. However, because @@TRANCOUNT is still 0, effectively there is no open transaction and therefore the message about no corresponding BEGIN TRANSACTION displays.

This situation highlights a potential problem with nested transactions in stored procedures. If you want each stored procedure to roll back only its own work if it encounters an error, you should test for an existing transaction, skip the step to begin a new transaction if one exists, and use a savepoint to roll back the to the start of the current transaction if an error occurs in the stored procedure. (We discuss savepoints in more detail in the next section.) Furthermore, the COMMIT statement in the stored procedure should execute only if the stored procedure starts its own transaction. By storing the @@TRANCOUNT value in a variable before you execute the remaining stored procedure's statements, you can later test whether a transaction existed at the start. If it did not, the variable's value is 0 and you can then safely commit the transaction that the stored procedure started. If a transaction did exist, no further action is required in the stored procedure.

We can revise the previous example to avoid nesting transactions as shown in Listing 3-3.

LISTING 3-3 Create a stored procedure that avoids a nested transaction

Click here to view code image

```
CREATE PROCEDURE Examples.DeleteParentNoNest
    @ParentId INT
AS
    DECLARE @CurrentTranCount INT;
    SELECT @CurrentTranCount = @@TRANCOUNT;
```

```
    IF (@CurrentTranCount = 0)
        BEGIN TRANSACTION DeleteTran;
    ELSE
        SAVE TRANSACTION DeleteTran;
    DELETE Examples.TestParent
    WHERE ParentId = @ParentId;
    IF @@ERROR != 0
        BEGIN
            ROLLBACK TRANSACTION DeleteTran;
            RETURN;
        END
    IF (@CurrentTranCount = 0)
        COMMIT TRANSACTION;
GO
BEGIN TRANSACTION;
    INSERT INTO Examples.TestParent(ParentId, ParentName)
    VALUES (7, 'Mary');
    EXEC Examples.DeleteParentNoNest @ParentId=3;
IF @@ERROR != 0
    BEGIN
        ROLLBACK TRANSACTION;
    RETURN
END
COMMIT TRANSACTION;
GO
```

When you execute the statements in Listing 3-3 and then check the table, you find that the new row is committed in the table and the row with the ParentId value of 3 remains in the table because the foreign key constraint caused SQL Server to roll back that transaction.

```
ParentId   ParentName
---------- ------------
1          Dean
2          Michael
3          Bob
4          Linda
5          Isabelle
6          Lukas
7          Mary
```

Exam Tip

Be sure that you understand when SQL Server increments and decrements @@TRANCOUNT and how to implement error handling for transactions.

The explicit transactions described to this point are all local transactions. Another option is to execute a distributed transaction when you need to execute statements on more than one server. To do this, start the transaction with the BEGIN DISTRIBUTED TRANSACTION and then end it with either COMMIT TRANSACTION or ROLLBACK TRANSACTION statements. The server on which you execute the distributed transaction controls the completion of the transaction.

Implement savepoints within transactions

A *savepoint* is a named location from which a transaction can restart if part of it is conditionally canceled. That means you can rollback a transaction to a specific savepoint if a statement does not complete successfully, as shown in the previous example.

When you assign a savepoint name, you should use 32 characters or less. SQL Server allows you to assign a longer name, but the statement uses only the first 32 characters. Bear in mind that the savepoint name is case-sensitive even if SQL Server is not configured for case sensitivity. Another option is to use a variable in the SAVE TRANSACTION statement, but the data type must be char, varchar, nchar, or nvarchar. If you use the same savepoint name multiple times in the same transaction, the ROLLBACK TRANSACTION statement rolls back to the most recent savepoint.

Normally, a ROLLBACK TRANSACTION resets the value of @@TRANCOUNT to 0. However, when a transaction rolls back to a savepoint, @@TRANCOUNT is not reset. The SAVE TRANSACTION statement also has no effect on @@TRANCOUNT.

In Listing 3-4, the transaction has multiple savepoints and SELECT statements illustrate the effect of modifying data, and then rolling back to a specific savepoint.

Listing 3-4 Create a transaction with multiple savepoints

Click here to view code image

```
BEGIN TRANSACTION;
    INSERT INTO Examples.TestParent(ParentId, ParentName)
    VALUES (8, 'Ed');
    SAVE TRANSACTION StartTran;

    SELECT 'StartTran' AS Status, ParentId, ParentName
    FROM Examples.TestParent;

    DELETE Examples.TestParent
        WHERE ParentId = 7;
    SAVE TRANSACTION DeleteTran;

    SELECT 'Delete 1' AS Status, ParentId, ParentName
    FROM Examples.TestParent;
```

```
    DELETE Examples.TestParent
        WHERE ParentId = 6;
    SELECT 'Delete 2' AS Status, ParentId, ParentName
    FROM Examples.TestParent;

    ROLLBACK TRANSACTION DeleteTran;
    SELECT 'RollbackDelete2' AS Status, ParentId, ParentName
    FROM Examples.TestParent;

    ROLLBACK TRANSACTION StartTran;
    SELECT @@TRANCOUNT AS 'TranCount';
    SELECT 'RollbackStart' AS Status, ParentId, ParentName
    FROM Examples.TestParent;
COMMIT TRANSACTION;
GO
```

The queries interspersed throughout this transaction give us visibility into the behavior of the savepoint and roll back operations:

Click here to view code image

```
Status    ParentId    ParentName
--------- ----------- ----------------------------------------
------------------------
StartTran 1           Dean
StartTran 2           Mike
StartTran 3           Bob
StartTran 4           Linda
StartTran 5           Isabelle
StartTran 6           Lukas
StartTran 7           Mary
StartTran 8           Ed

Status   ParentId    ParentName
-------- ----------- ----------------------------------------
------------------------
Delete 1 1           Dean
Delete 1 2           Mike
Delete 1 3           Bob
Delete 1 4           Linda
Delete 1 5           Isabelle
Delete 1 6           Lukas
Delete 1 8           Ed

Status   ParentId    ParentName
-------- ----------- ----------------------------------------
------------------------
```

```
Delete 2 1           Dean
Delete 2 2           Mike
Delete 2 3           Bob
Delete 2 4           Linda
Delete 2 5           Isabelle
Delete 2 8           Ed

Status           ParentId    ParentName
---------------- ----------- ------------------------------------
------------------------
RollbackDelete2 1            Dean
RollbackDelete2 2            Mike
RollbackDelete2 3            Bob
RollbackDelete2 4            Linda
RollbackDelete2 5            Isabelle
RollbackDelete2 6            Lukas
RollbackDelete2 8            Ed

TranCount
-----------
1

Status         ParentId    ParentName
-------------- ----------- ------------------------------------
-------------------------
RollbackStart 1            Dean
RollbackStart 2            Mike
RollbackStart 3            Bob
RollbackStart 4            Linda
RollbackStart 5            Isabelle
RollbackStart 6            Lukas
RollbackStart 7            Mary
RollbackStart 8            Ed
```

The eight rows in the query with status StartTran show the condition of the table after the INSERT operation and reflects the state of the data for the StartTran savepoint. Next, the seven rows in the query with status *Delete 1* include one less row due to the DELETE operation. The DeleteTran savepoint includes this version of the table. After another DELETE operation executes, the query with status Delete 2 returns six rows. The first ROLLBACK TRANSACTION statement restores the version of data for the DeleteTran savepoint, and the query with status RollbackDelete2 correctly shows the seven rows prior to the second DELETE operation. Next, we can see that the @@TRANCOUNT variable at this point is still 1 because the ROLLBACK TRANSACTION statement did not reset it to 0. Last, another ROLLBACK TRANSACTION returns the table to its earlier state, which is committed at the end of the transaction.

Note Savepoints in distributed transactions

You cannot use savepoints in a distributed transaction beginning from an explicit BEGIN DISTRIBUTED TRANSACTION statement or a local transaction escalation.

Determine the role of transactions in high-concurrency databases

A high concurrency database should support a high number of simultaneous processes that do not interfere with one another while preserving the consistency of the data affected by those processes. Processes modifying data can potentially adversely affect processes trying to read or change the same data at the same time. To prevent simultaneous attempts to change the same data, SQL Server acquires locks for the current transaction, thereby blocking all other transactions.

Potential problems with concurrent processes

A failure to control concurrency in database can result in a variety of side effects. Typically, you want to design applications that avoid these problems. In some cases, your business requirements might allow a behavior. For now, let's focus on which potential problems might arise. In Skill 3.2, we explain how to use isolation levels to manage the behavior of concurrent transactions.

Dirty reads

A *dirty read*, also known as an uncommitted dependency, can occur when an uncommitted transaction updates a row at the same time that another transaction reads that row with its new value. Because the writing transaction is not committed, the row could revert to its original state and consequently the reading transaction has data that is not valid.

SQL Server does not allow dirty reads by default. However, by controlling the isolation level of the reading transaction, you can specify whether it reads both uncommitted and committed data or committed data only.

Non-repeatable reads

A *non-repeatable read* can occur when data is read more than once within the same transaction while another transaction updates the same data between read operations. Let's say that a transaction reads the current in-stock quantity of a widget from an inventory table as 5 and continues to perform other operations, which leaves the transaction in an uncommitted state. During this time, another transaction changes the in-stock quantity of the widget to 3. Then the first transaction reads the in-stock quantity of the widget again, which is now inconsistent with the initial value read.

Phantom reads

Closely related to a non-repeatable read is a phantom read. This potential problem can occur when one transaction reads the same data multiple times while another transaction inserts or updates a row between read operations. As an example, consider a transaction in which a SELECT statement reads rows having in-stock quantities less than 5 from the inventory table and remains uncommitted while a second transaction inserts a row with an in-stock quantity of 1. When the first transaction reads the inventory table again, the number of rows increases by one. In this case, the additional row is considered to be a phantom row. This situation occurs only when the query uses a predicate.

Lost updates

Another potential problem can occur when two processes read the same row and then update that data with different values. This might happen if a transaction first reads a value into a variable and then uses the variable in an update statement in a later step. When this update executes, another transaction updates the same data. Whichever of these transactions is committed first becomes a *lost update* because it was replaced by the update in the other transaction. You cannot use isolation levels to change this behavior, but you can write an application that specifically allows lost updates.

Resource locks

SQL Server locks the minimum number of resources required to complete a transaction. It uses different types of locks to support as much concurrency as possible while maintaining data consistency and transaction isolation. The SQL Server Lock Manager chooses the lock mode and resources to lock based on the operation to be performed, the amount of data to be affected by the operation, and the isolation level type (described in Skill 3.2). It also manages the compatibility of locks on the same resources, resolves deadlocks when possible, and escalates locks when necessary (as described in Skill 3.3).

SQL Server takes locks on resources at several levels to provide the necessary protection for a transaction. This group of locks at varying levels of granularity is known as a *lock hierarchy* and consists of one or more of the following lock modes:

- **Shared (S)** This lock mode, also known as a read lock, is used for SELECT, INSERT, UPDATE, and DELETE operations and is released as soon as data has been read from the locked resource. While the resource is locked, other transactions cannot change its data. However, in theory, an unlimited number of shared (s) locks can exist on a resource simultaneously. You can force SQL Server to hold the lock for the duration of the transaction by adding the HOLDLOCK table hint like this:

Click here to view code image

```
BEGIN TRANSACTION;
SELECT ParentId, ParentName
FROM Examples.TestParent WITH (HOLDLOCK);
WAITFOR DELAY '00:00:15';
```

```
ROLLBACK TRANSACTION;
```

Another way to change the lock's duration is to set the `REPEATABLE_READ` or `SERIALIZABLE` transaction isolation levels, which we explain in more detail in Skill 3.2.

- **Update (U)** SQL Server takes this lock on a resource that might be updated in order to prevent a common type of deadlocking, which we describe further in Skill 3.3. Only one update (U) lock can exist on a resource at a time. When a transaction modifies the resource, SQL Server converts the update (U) lock to an exclusive (X) lock.
- **Exclusive (X)** This lock mode protects a resource during INSERT, UPDATE, or DELETE operations to prevent that resource from multiple concurrent changes. While the lock is held, no other transaction can read or modify the data, unless a statement uses the NOLOCK hint or a transaction runs under the read uncommitted isolation level as we describe in Skill 3.2
- **Intent** An intent lock establishes a lock hierarchy to protect a resource at a lower level from getting a shared (S) lock or exclusive (X) lock. Technically speaking, intent locks are not true locks, but rather serve as an indicator that actual locks exist at a lower level. That way, another transaction cannot try to acquire a lock at the higher level that is incompatible with the existing lock at the lower level. There are six types of intent locks:
  - **Intent shared (IS)** With this lock mode, SQL Server protects requested or acquired shared (S) locks on some resources lower in the lock hierarchy.
  - **Intent exclusive (IX)** This lock mode is a superset of intent shared (IS) locks that not only protects locks on resources lower in the hierarchy, but also protects requested or acquired exclusive (X) locks on some resources lower in the hierarchy.
  - **Shared with intent exclusive (SIX)** This lock mode protects requested or acquired shared (S) locks on all resources lower in the hierarchy and intent exclusive (IX) locks on some resources lower in the hierarchy. Only one shared with intent exclusive (SIX) lock can exist at a time for a resource to prevent other transactions from modifying it. However, lower level resources can have intent shared (IS) locks and can be read by other transactions.
  - **Intent update (IU)** SQL Server uses this lock mode on page resources only to protect requested or acquired update (U) locks on all lower-level resources and converts it to an intent exclusive (IX) lock if a transaction performs an update operation.
  - **Shared intent update (SIU)** This lock mode is a combination of shared (S) and intent update (IU) locks and occurs when a transaction acquires each lock separately but holds them at the same time.

- **Update intent exclusive (UIX)** This lock mode results from a combination of update (U) and intent exclusive (IX) locks that a transaction acquires separately but holds at the same time.

- **Schema** SQL Server acquires this lock when an operation depends the table's schema. There are two types of schema locks:
  - **Schema modification (Sch-M)** This lock mode prevents other transactions from reading from or writing to a table during a Data Definition Language (DDL) operation, such as removing a column. Some Data Manipulation Language (DML) operations, such as truncating a table, also require a schema modification (Sch-M) lock.
  - **Schema stability (Sch-S)** SQL Server uses this lock mode during query compilation and execution to block concurrent DDL operations and concurrent DML operations requiring a schema modification (Sch-M) lock from accessing a table.
- **Bulk Update (BU)** This lock mode is used for bulk copy operations to allow multiple threads to bulk load data into the same table at the same time and to prevent other transactions that are not bulk loading data from accessing the table. SQL Server acquires it when the table lock on bulk load table option is set by using sp_tableoption or when you use a TABLOCK hint like this:

Click here to view code image

```
INSERT INTO Examples.TestParent WITH (TABLOCK)
SELECT <columns> FROM <table>;
```

- **Key-range** A key-range lock is applied to a range of rows that is read by a query with the SERIALIZABLE isolation level to prevent other transactions from inserting rows that would be returned in the serializable transaction if the same query executes again. In other words, this lock mode prevents phantom reads within the set of rows that the transaction reads.
  - **RangeS-S** This lock mode is a shared range, shared resource lock used for a serializable range scan.
  - **RangeS-U** This lock mode is a shared range, update resource lock used for a serializable update scan.
  - **RangeI-N** This lock mode is an insert range, null resource lock that SQL Server acquires to test a range before inserting a new key into an index.
  - **RangeX-X** This lock mode is an exclusive range, exclusive resource lock used when updating a key in a range.

While many locks are compatible with each other, some locks prevent other transactions from acquiring locks on the same resource, as shown in Table 3-1. Let's consider a situation in which one transaction has a shared (S) lock on a row and another transaction is

requesting an exclusive (X) lock. In this case, the request is blocked until the first transaction releases its lock.

	Existing granted mode					
Requested mode	**S**	**U**	**X**	**IS**	**IX**	**SIX**
S	Yes	Yes	No	Yes	No	No
U	Yes	No	No	Yes	No	No
X	No	No	No	No	No	No
IS	Yes	Yes	No	Yes	Yes	Yes
IX	No	No	No	Yes	Yes	No
SIX	No	No	No	Yes	No	No

TABLE 3-1 Lock compatibility for commonly encountered lock modes

> **Need More Review? Lock compatibility**
>
> For a complete matrix of lock compatibility, see "Lock Compatibility (Database Engine)" at *https://technet.microsoft.com/en-us/library/ms186396(v=sql.105).aspx*.

SQL Server can acquire a lock on any of the following resources to ensure that the user of that resource has a consistent view of the data throughout a transaction:

- **RID** A row identifier for the single row to lock within a heap and is acquired when possible to provide the highest possible concurrency.
- **KEY** A key or range of keys in an index for a serializable transaction can be locked in one of two ways depending on the isolation level. If a transaction runs in the READ COMMITTED or REPEATABLE READ isolation level, the index keys of the accessed rows are locked. If the table has a clustered index, SQL Server acquires key locks instead of row locks because the data rows are the leaf-level of the index. If a transaction runs in the SERIALIZABLE isolation mode, SQL Server acquires key-range locks to prevent phantom reads.
- **PAGE** An 8-kilobyte (KB) data or index page gets locked when a transaction reads all rows on a page or when page-level maintenance, such as updating page pointers after a page-split, is performed.
- **EXTENT** A contiguous block of eight data or index pages gets a shared (S) or exclusive (X) locks typically during space allocation and de-allocation.
- **HoBT** A heap or B-Tree lock can be an entire index or all data pages of a heap.
- **Table** An entire table, including both data and indexes, can be locked for SELECT, UPDATE, or DELETE operations.

- **File** A database file can be locked individually.
- **Application** A resource defined by your application can be locked by using sp_getapplock so that you can lock any resource you want with a specified lock mode.
- **Metadata** Any system metadata can be locked to protect system catalog information.
- **Allocation unit** An database allocation unit used for storage of data can be locked.
- **Database** An entire database gets a shared (S) lock to indicate it is currently in use so that another process cannot drop it, take it offline, or restore it.

To increase concurrency, SQL Server uses dynamic lock management. That is, in a large table for which many row locks are required (as determined by the query optimizer), SQL Server might instead take a page or table lock at the beginning of a transaction. SQL Server can also escalate lock modes dynamically during a transaction. For example, if the transaction initially has a set of row locks, and later requests more row locks, SQL Server releases the row locks and takes a table lock. This behavior simplifies lock management, but reduces concurrency.

Exam Tip

Locks and lock escalation in SQL Server are important concepts covered in the exam that you should understand thoroughly.

Note Implicit transaction locks

Be aware that when you use implicit transactions, SQL Server holds locks until you commit the transaction. This behavior can reduce concurrency and interfere with truncation of the transaction log.

Skill 3.2: Manage isolation levels

SQL Server uses isolation levels to manage conflict between two transactions attempting to use or change the same data at the same time. Furthermore, because the way in which you implement transactions impacts database performance, you need to understand the differences between isolation levels and be familiar with the scenarios with which each is best suited. Given a scenario in which an isolation level and a set of concurrent queries are specified, you should be able to predict the outcome of the queries. In addition, you should understand the types of locks that SQL Server acquires for each isolation level, if applicable, as well as the effect on other resources, such as *tempdb*, and the resulting potential performance impact of using a specific isolation level.

This section covers how to:

- Identify differences between isolation levels
- Define results of concurrent queries based on isolation level
- Identify the resource and performance impact of given isolation levels

Identify differences between isolation levels

At one end of the spectrum, SQL Server can protect data completely to prevent one transaction from seeing the effects of another transaction, while at the other end of the spectrum, it can give all transactions full access to the data. It does this by using isolation levels to control whether a lock is acquired during a read, the type of lock, and the duration of the lock. Isolation levels also determine whether a read operation can access rows that have been changed by another transaction and whether it can access uncommitted rows. Additionally, isolation levels block transactions requiring access to a resource with an exclusive lock.

It is important to note that setting an isolation level does not change the way in which SQL Server acquires locks. If a transaction modifies data, SQL Server always acquires an exclusive (X) lock on the data to change, and holds the lock for the duration of the transaction. The purpose of the isolation levels is to specify how read operations should behave when other concurrent transactions are changing data.

If you lower the isolation level, you can increase the number of concurrent transactions that SQL Server processes, but you also increase the risk of dirty reads and other problems associated with concurrent processes as we described in Skill 3.1. If you raise the isolation level, you minimize these concurrency problems, but transactions are more likely to block one another and performance is more likely to suffer. Therefore, you must find the appropriate balance between protecting data and the effect of each isolation level.

SQL Server supports both pessimistic and optimistic isolation levels for concurrency management. Pessimistic isolation levels use blocking to avoid conflicts whereas optimistic isolation levels use snapshots of the data to enable higher concurrency. Pessimistic isolation levels rely on locks to prevent changes to data during read operations and to block read operations on data that is being changed by another operation. Optimistic isolation levels make a copy of data for read operations so that write operations can proceed unhindered. If SQL Server detects two write operations attempting to modify the same data at the same time, it returns a message to the application in which there should be appropriate logic for resolving this conflict.

Exam Tip

It is important to understand the differences between SQL Server isolation levels and scenarios for which each is appropriate.

Read Committed

READ COMMITTED is the default isolation level for SQL Server. It uses pessimistic locking to protect data. With this isolation level set, a transaction cannot read uncommitted data that is being added or changed by another transaction. A transaction attempting to read data that is currently being changed is blocked until the transaction changing the data releases the lock. A transaction running under this isolation level issues shared locks, but releases row or page locks after reading a row. If your query scans an index while another transactions changes the index key column of a row, that row could appear twice in the query results if that key change moved the row to a new position ahead of the scan. Another option is that it might not appear at all if the row moved to a position already read by the scan.

Read Uncommitted

The READ UNCOMMITTED isolation level is the least restrictive setting. It allows a transaction to read data that has not yet been committed by other transactions. SQL Server ignores any locks and reads data from memory. Furthermore, transactions running under this isolation level do not acquire shared (S) locks to prevent other transactions from changing the data being read. Last, if a transaction is reading rows using an allocation order scan when another transaction causes a page split, your query can miss rows. For these reasons, READ UNCOMMITTED is never a good choice for line of business applications where accuracy matters most, but might be acceptable for a reporting application where the performance benefit outweighs the need for a precise value.

Repeatable Read

When you set the REPEATABLE READ isolation level, you ensure that any data read by one transaction is not changed by another transaction. That way, the transaction can repeat a query and get identical results each time. In this case, the data is protected by shared (S) locks. It is important to note that the only data protected is the existing data that has been read. If another transaction inserts a new row, the first transaction's repeat of its query could return this row as a phantom read.

Serializable

The most pessimistic isolation level is SERIALIZABLE, which uses range locks on the data to not only prevent changes but also insertions. Therefore, phantom reads are not possible when you set this isolation level. Each transaction is completely isolated from one another even when they execute in parallel or overlap.

Snapshot

The SNAPSHOT isolation level is optimistic and allows read and write operations to run concurrently without blocking one another. Unlike the other isolation levels, you must first

configure the database to allow it, and then you can set the isolation level for a transaction. As long as a transaction is open, SQL Server preserves the state of committed data at the start of the transaction and stores any changes to the data by other transactions in *tempdb*. It increases concurrency by eliminating the need for locks for read operations.

> **Note SNAPSHOT isolation and distributed transactions**
>
> You cannot use SNAPSHOT isolation with distributed transactions. In addition, you cannot use enable it in the following databases: master, msdb, and tempdb.

Read Committed Snapshot

The READ_COMMITTED_SNAPSHOT isolation level is an optimistic alternative to READ COMMITTED. Like the SNAPSHOT isolation level, you must first enable it at the database level before setting it for a transaction. Unlike SNAPSHOT isolation, you can use the READ_COMMITTED_SNAPSHOT isolation level with distributed transactions. The key difference between the two isolation levels is the ability with READ_COMMITTED_SNAPSHOT for a transaction to repeatedly read data as it was at the start of the read statement rather than at the start of the transaction. When each statement executes within a transaction, SQL Server takes a new snapshot that remains consistent until the next statement executes.

You use this isolation level when your application executes a long-running, multi-statement query and requires the data to be consistent to the point in time that the query starts. You should also consider using this isolation level when enough read and write blocking occurs that the resource overhead of maintaining row snapshots is preferable and there is little likelihood of a transaction rolling back due to an update conflict.

Define results of concurrent queries based on isolation level

To better appreciate the effect of concurrent queries, let's consider a scenario that involves two users that are operating on the same data. One user starts executing a query that results in a full table scan and normally takes several minutes to complete. Meanwhile, a minute after the read operation begins, the other user updates and commits row in the same table that has not yet been read by the first user's query. The rows returned by the first user's query depend on the isolation levels set for that user.

Before we look at each isolation level's effect on this scenario, let's create a table and add some data as shown in Listing 3-5.

LISTING 3-5 Create a test environment for testing isolation levels

Click here to view code image

```
CREATE TABLE Examples.IsolationLevels
(
    RowId  int NOT NULL
        CONSTRAINT PKRowId PRIMARY KEY,
    ColumnText  varchar(100) NOT NULL
);

INSERT INTO Examples.IsolationLevels(RowId, ColumnText)
VALUES (1, 'Row 1'), (2, 'Row 2'), (3, 'Row 3'), (4, 'Row 4');
```

You use the SET TRANSACTION ISOLATION LEVEL statement when you want to override the default isolation level and thereby change the way a SELECT statement behaves with respect to other concurrent operations. It is important to know that this statement changes the isolation level for the user session. If you want to change the isolation level for a single statement only, use a table hint instead.

Read Committed

Because this isolation level only reads committed data, dirty reads are prevented. However, if query reads the same data multiple times, non-repeatable reads or phantom reads are possible.

Because the READ COMMITTED isolation level is the default, you do not need to explicitly set the isolation level. However, if you had previously changed the isolation level for the user session or the database, you can revert it to the default isolation level by executing the following statement:

Click here to view code image

```
SET TRANSACTION ISOLATION LEVEL READ COMMITTED;
```

To test the behavior of the READ COMMITTED isolation level, execute the following statements:

Click here to view code image

```
BEGIN TRANSACTION;
    UPDATE Examples.IsolationLevels
        SET ColumnText = 'Row 1 Updated'
        WHERE RowId = 1;
```

In a new session, read the table that you just updated:

Click here to view code image

```
SELECT RowId, ColumnText
FROM Examples.IsolationLevels;
```

In this case, the update operation blocks the read operations. Return to the first session

and restore the data by rolling back the transaction:

```
ROLLBACK TRANSACTION;
```

Now the second session's read request completes successfully, and the results do not include the updated row because it was never committed.

```
RowId   ColumnText
------- ------------
1       Row 1
2       Row 2
3       Row 3
4       Row 4
```

Read Uncommitted

This isolation level allows dirty reads, non-repeatable reads, and phantom reads. On the other hand, a transaction set to this isolation level executes quickly because locks and validations are ignored.

Let's observe this behavior by starting a transaction without committing it:

Click here to view code image

```
BEGIN TRANSACTION;
    UPDATE Examples.IsolationLevels
        SET ColumnText = 'Row 1 Updated'
        WHERE RowId = 1;
```

Now open a new session, change the isolation level, and read the table that you just updated:

Click here to view code image

```
SET TRANSACTION ISOLATION LEVEL READ UNCOMMITTED;
SELECT RowId, ColumnText
FROM Examples.IsolationLevels;
```

The results include the updated row:

```
RowId   ColumnText
------- ------------
1       Row 1 Updated
2       Row 2
3       Row 3
4       Row 4
```

Return to the first session and roll back the transaction:

```
ROLLBACK TRANSACTION;
```

Then in the second session, read the table again:

Click here to view code image

```
SELECT RowId, ColumnText
FROM Examples.IsolationLevels;
```

Now the results show the data in its state prior to the update that rolled back:

```
RowId   ColumnText
------- ------------
1       Row 1
2       Row 2
3       Row 3
4       Row 4
```

Rather than change the isolation level at the session level, you can force the read uncommitted isolation level by using the NOLOCK hint. Repeat the previous example by using two new sessions to revert to the default isolation level and replacing the statements in the second session with the following statement:

Click here to view code image

```
SELECT RowId, ColumnText
FROM Examples.IsolationLevels
WITH (NOLOCK);
```

Repeatable Read

The behavior of the REPEATABLE READ isolation level is much like that of READ COMMITTED, except that it ensures that multiple reads of the same data within a transaction is consistent. Dirty reads and non-repeatable reads are prevented, although phantom reads are a possible side effect because range locks are not used.

We can see the effects of using REPEATABLE READ by running statements in separate sessions. Start by adding the following statements in one new session:

Click here to view code image

```
SET TRANSACTION ISOLATION LEVEL REPEATABLE READ;
BEGIN TRANSACTION;
    SELECT RowId, ColumnText
    FROM Examples.IsolationLevels;
    WAITFOR DELAY '00:00:15';
    SELECT RowId, ColumnText
    FROM Examples.IsolationLevels;
ROLLBACK TRANSACTION;
```

In the second session, add the following statements and then with both sessions visible, execute both sessions:

Click here to view code image

```
UPDATE Examples.IsolationLevels
    SET ColumnText = 'Row 1 Updated'
    WHERE RowId = 1;
```

In this case, the first read operations blocks the update operation, which executes when the first read's locks are released, the update commits the data change, but the second query returns the same rows as the first query due to the isolation level of the transaction:

```
RowId   ColumnText
------- ------------
1       Row 1
2       Row 2
3       Row 3
4       Row 4

RowId   ColumnText
------- ------------
1       Row 1
2       Row 2
3       Row 3
4       Row 4
```

If you check the table values again, you can see that the updated row appears in the query results:

```
RowId   ColumnText
------- ------------
1       Row 1 Updated
2       Row 2
3       Row 3
4       Row 4
```

Serializable

The SERIALIZABLE isolation level behaves like REPEATABLE READ, but goes one step further by ensuring new rows added after the start of the transaction are not visible to the transaction's statement. Therefore, dirty reads, non-repeatable reads, and phantom reads are prevented.

Before we see how the SERIALIZABLE isolation level works, let's look at an example that produces a phantom read. In one new session, add the following statements:

Click here to view code image

```
SET TRANSACTION ISOLATION LEVEL REPEATABLE READ;
BEGIN TRANSACTION;
    SELECT RowId, ColumnText
    FROM Examples.IsolationLevels;
    WAITFOR DELAY '00:00:15';
```

```
    SELECT RowId, ColumnText
    FROM Examples.IsolationLevels;
ROLLBACK TRANSACTION;
```

As in the previous examples, start a new session to insert a row into the same table, and execute both sessions:

Click here to view code image

```
INSERT INTO Examples.IsolationLevels(RowId, ColumnText)
VALUES (5, 'Row 5');
```

In this case, the transaction starts with a read operation and returns four rows, but does not block the insert operation. The REPEATABLE READ isolation level only prevents changes to data that has been read, but does not prevent the transaction from seeing the new row, which is returned by the second query as shown here:

```
RowId   ColumnText
------- ------------
1       Row 1 Updated
2       Row 2
3       Row 3
4       Row 4

RowId   ColumnText
------- ------------
1       Row 1 Updated
2       Row 2
3       Row 3
4       Row 4
5       Row 5
```

Replace the isolation level statement in the first session with this statement to change the isolation level:

Click here to view code image

```
SET TRANSACTION ISOLATION LEVEL SERIALIZABLE;
```

Then create a new session to insert another row:

Click here to view code image

```
INSERT INTO Examples.IsolationLevels(RowId, ColumnText)
VALUES (6, 'Row 6');
```

This time because the INSERT operation is blocked by the transaction, both queries return the same results without the new row.

```
RowId   ColumnText
------- ------------
```

```
1       Row 1
2       Row 2
3       Row 3
4       Row 4
5       Row 5

RowId   ColumnText
------- ------------
1       Row 1
2       Row 2
3       Row 3
4       Row 4
5       Row 5
```

After the transaction ends, any subsequent queries to the table return six rows. The trade-off for this consistency during the transaction is the blocking of write operations.

Snapshot

Snapshot Isolation gives you the same data for the duration of the transaction. This level of protection prevents dirty reads, non-repeatable reads, and phantom reads. As other transactions update or delete rows, a copy of the modified row is inserted into *tempdb*. This row also includes a transaction sequence number so that SQL Server can determine which version to use for a new transaction's snapshot. When the new transaction executes a read request, SQL Server scans the version chain to find the latest committed row having a transaction sequence number lower than the current transaction. Periodically, SQL Server deletes row versions for transactions that are no longer open.

To use the SNAPSHOT isolation level, you must first enable it at the database level by using the following statement:

Click here to view code image

```
ALTER DATABASE ExamBook762Ch3
SET ALLOW_SNAPSHOT_ISOLATION ON;
```

Now set the isolation level for the session and start a transaction:

Click here to view code image

```
SET TRANSACTION ISOLATION LEVEL SNAPSHOT;
BEGIN TRANSACTION;
    SELECT RowId, ColumnText
    FROM Examples.IsolationLevels;
    WAITFOR DELAY '00:00:15';
    SELECT RowId, ColumnText
    FROM Examples.IsolationLevels;
ROLLBACK TRANSACTION;
```

Then set up a write operation in a new second session:

Click here to view code image

```
INSERT INTO Examples.IsolationLevels(RowId, ColumnText)
VALUES (7, 'Row 7');
```

The write operation runs immediately because it is no longer blocked by the read operations, yet the query results return only the six rows that existed prior to the insertion.

> **Note SNAPSHOT isolation and tempdb**
>
> If you access global temp tables within a transaction set to SNAPSHOT isolation, you must first enable the ALLOW_SNAPSHOT_ISOLATION database option for *tempdb*. As an alternative, you can use a hint to change the isolation level for the statement.

If you have a transaction that reads from a database that is enabled for SNAPSHOT isolation and another database that is not enabled, the transaction fails. To execute successfully, the transaction must include a table hint for the database without SNAPSHOT isolation level enabled.

Let's set up another database and a new table as shown in Listing 3-6.

LISTING 3-6 Create a separate for testing isolation levels

Click here to view code image

```
CREATE DATABASE ExamBook762Ch3_IsolationTest;
GO
USE ExamBook762Ch3_IsolationTest;
GO
CREATE SCHEMA Examples;
GO
CREATE TABLE Examples.IsolationLevelsTest
(RowId INT NOT NULL
    CONSTRAINT PKRowId PRIMARY KEY,
    ColumnText  varchar(100) NOT NULL
);
INSERT INTO Examples.IsolationLevelsTest(RowId, ColumnText)
VALUES (1, 'Row 1'), (2, 'Row 2'), (3, 'Row 3'), (4, 'Row 4');
```

Now try to execute the following transaction that joins the data from the snapshot-enabled database with data from the other database:

Click here to view code image

```
SET TRANSACTION ISOLATION LEVEL SNAPSHOT;
BEGIN TRANSACTION;
    SELECT t1.RowId, t2.ColumnText
    FROM Examples.IsolationLevels AS t1
    INNER JOIN
ExamBook762Ch3_IsolationTest.Examples.IsolationLevelsTest AS t2
    ON t1.RowId = t2.RowId;
END TRANSACTION;
```

SQL Server returns the following error:

Click here to view code image

```
Msg 3952, Level 16, State 1, Line 5
Snapshot isolation transaction failed accessing database
'ExamBook762Ch3_IsolationTest'
because snapshot isolation is not allowed in this database. Use
ALTER DATABASE to allow
snapshot isolation.
```

You might not always have the option to alter the other database to enable Snapshot isolation. Instead, you can change the isolation level of the transaction's statement to READ COMMITTED, which allows the transaction to execute successfully:

Click here to view code image

```
SET TRANSACTION ISOLATION LEVEL SNAPSHOT;
BEGIN TRANSACTION;
    SELECT t1.RowId, t2.ColumnText
    FROM Examples.IsolationLevels AS t1
    INNER JOIN
ExamBook762Ch3_IsolationTest.Examples.IsolationLevelsTest AS t2
    WITH (READCOMMITTED)
    ON t1.RowId = t2.RowId;
END TRANSACTION;
```

Another problem that you might encounter when using this isolation level is an update conflict, which causes the transaction to terminate and roll back. This situation can occur when one transaction using the SNAPSHOT isolation level reads data that another transaction modifies and then the first transaction attempts to update the same data. (This situation does not occur when a transaction runs using the READ_COMMITTED_SNAPSHOT isolation level.)

A problem can also arise when the state of the database changes during the transaction. As one example, a transaction set to SNAPSHOT isolation fails when the database is changed to read-only after the transaction starts, but before it accesses the database. Likewise, a failure occurs if a database recovery occurred in that same interval. A database recovery can be caused when the database is set to OFFLINE and then to ONLINE, when it auto-closes and re-opens, or when an operation detaches and attaches the

database.

It is important to know that row versioning applies only to data and not to system metadata. If a statement changes metadata of an object while a transaction using the SNAPSHOT isolation level is open and the transaction subsequently references the modified object, the transaction fails. Be aware that BULK INSERT operations can change a table's metadata and cause transaction failures as a result. (This behavior does not occur when using the READ_COMMITTED_SNAPSHOT isolation level.)

One way to see this behavior is to change an index on a table while a transaction is open. Let's first add an index to a table:

Click here to view code image

```
CREATE INDEX Ix_RowId ON Examples.IsolationLevels (RowId);
```

Next set up a new transaction:

Click here to view code image

```
SET TRANSACTION ISOLATION LEVEL SNAPSHOT;
BEGIN TRANSACTION;
    SELECT RowId, ColumnText
    FROM Examples.IsolationLevels;
    WAITFOR DELAY '00:00:15';
    SELECT RowId, ColumnText
    FROM Examples.IsolationLevels;
ROLLBACK TRANSACTION;
```

Then set up a second session to change the index by using the following statement and execute both sessions:

Click here to view code image

```
ALTER INDEX Ix_RowId
    ON Examples.IsolationLevels REBUILD;
```

SQL Server returns the following error due to the metadata change:

Click here to view code image

```
Msg 3961, Level 16, State 1, Line 6
Snapshot isolation transaction failed in database
'ExamBook762Ch3' because the object
accessed by the statement has been modified by a DDL statement
in another concurrent
transaction since the start of this transaction.  It is
disallowed because the metadata
is not versioned. A concurrent update to metadata can lead to
inconsistency if mixed
with snapshot isolation.
```

Be sure to disable snapshot isolation after completing the examples in this section:

Click here to view code image

```
ALTER DATABASE ExamBook762Ch3
SET ALLOW_SNAPSHOT_ISOLATION OFF;
```

Read Committed snapshot

To use the READ_COMMITTED_SNAPSHOT isolation level, you need only enable it at the database level by using the following statement:

Click here to view code image

```
ALTER DATABASE ExamBook762Ch3
SET READ_COMMITTED_SNAPSHOT ON;
```

With this setting enabled, all queries that normally execute using the READ COMMITTED isolation level switch to using the READ_COMMITTED_SNAPSHOT isolation level without requiring you to change the query code. SQL Server creates a snapshot of committed data when each statement starts. Consequently, read operations at different points in a transaction might return different results.

During the transaction, SQL Server copies rows modified by other transactions into a collection of pages in tempdb known as the *version store*. When a row is updated multiple times, a copy of each change is in the version store. This set of row versions is called a version chain.

Let's see how this isolation level differs from the SNAPSHOT isolation level by setting up a new session:

Click here to view code image

```
BEGIN TRANSACTION;
    SELECT RowId, ColumnText
    FROM Examples.IsolationLevels;
    WAITFOR DELAY '00:00:15';
    SELECT RowId, ColumnText
    FROM Examples.IsolationLevels;
ROLLBACK TRANSACTION;
```

Next, set up a write operation in a new second session, and then execute both sessions:

Click here to view code image

```
INSERT INTO Examples.IsolationLevels(RowId, ColumnText)
VALUES (8, 'Row 8');
```

Just as with the SNAPSHOT isolation level, the write operation runs immediately because read operations are not blocking it. However, each query returns different results because the statements read different versions of the data.

```
RowId   ColumnText
------- ------------
1       Row 1
2       Row 2
3       Row 3
4       Row 4
5       Row 5
6       Row 6
7       Row 7

RowId   ColumnText
------- ------------
1       Row 1
2       Row 2
3       Row 3
4       Row 4
5       Row 5
6       Row 6
7       Row 7
8       Row 8
```

Last, disable the READ_COMMITTED_SNAPSHOT isolation level after completing this example:

Click here to view code image

```
ALTER DATABASE Examples
SET READ_COMMITTED_SNAPSHOT OFF;
```

Identify the resource and performance impact of given isolation levels

The goal of isolation levels is to ensure that queries return complete and consistent results while other concurrent processes are running. To avoid locking contention and improve overall performance, you should keep each transaction short and concise so it can execute quickly while holding the fewest and smallest possible locks.

Read Committed

With this isolation level, SQL Server holds two types of locks. A shared (S) lock is acquired for read operations and is held only for the duration of that single operation. On the other hand, an exclusive (X) lock is acquired for a write operation. Any changes to the data are not visible to other operations for the duration of the write operation's transaction.

Read Uncommitted

SQL Server ignores existing locks and reads both committed and uncommitted data. Furthermore, it does not acquire shared locks for read operations. However, schema modification locks can still block reads.

Repeatable Read

SQL Server places Shared (S) locks on the data (and up the lock hierarchy) for the duration of the transaction. Therefore, reads block write operations in other transactions. Consequently, SQL Server cannot manage as many concurrent processes and performance can be adversely impacted as deadlocks can become more frequent.

Serializable

SQL Server locks data for a read operation and also uses key-range locks to prevent any other transactions from inserting or modifying the data for the duration of a transaction. This high level of locking reduces concurrency and potentially slows performance due to locking contention.

Snapshot

No locks are acquired for this isolation level. Consequently, deadlocks and lock escalations occur less frequently, performance is faster, and concurrency is higher. Read operations are not blocked by write operations, and write operations are not blocked by read operations.

On the other hand, these benefits come with an overhead cost. More space is required in tempdb for row version storage and more CPU and memory is required by SQL Server to manage row versioning. Update operations might run slower as a result of the extra steps required to manage row versions. Furthermore, long running read operations can run slower if many updates or deletes are occurring and increasing the length of the version chains that SQL Server must scan. You can improve performance by placing tempdb on a dedicated, high-performance disk drive.

> **Note SNAPSHOT isolation and tempdb disk space**
>
> When using this isolation level, it is important to make sure there is enough disk space for tempdb. If it runs out of space, update operations can complete successfully, but the read operations relying on row version might fail.

Read Committed Snapshot

When a new transaction using the READ_COMMITTED_SNAPSHOT isolation level requests locked data, SQL Server provides a copy of the data. It does not acquire shared page or row locks. As a consequence, reads do not block write operations and writes do not block read operations, although writes do require exclusive locks and continue to block other writes until the end of the transaction. However, because SQL Server removes row versions from tempdb when a transaction is over, it is possible to experience some concurrency side effects.

Note READ_COMMITTED_SNAPSHOT isolation and tempdb disk space

READ_COMMITTED_SNAPSHOT uses less tempdb space than snapshot isolation, but it is still important to ensure tempdb has enough space for both normal operations and row versioning. Note that both READ_COMMITTED_SNAPSHOT and SNAPSHOT isolation levels can be enabled at the same time, but there is only one copy of data in the version store.

Skill 3.3: Optimize concurrency and locking behavior

SQL Server uses locks to control the effect of concurrent transactions on one another. Part of your job as an administrator is to improve concurrency by properly managing locking behavior. That means you need to understand how to uncover performance problems related to locks and lock escalations. Additionally, you must know how to use the tools available to you for identifying when and why deadlocks happen and the possible steps you can take to prevent deadlocks from arising.

This section covers how to:

- Troubleshoot locking issues
- Identify lock escalation behaviors
- Capture and analyze deadlock graphs
- Identify ways to remediate deadlocks

Troubleshoot locking issues

Before you can troubleshoot locking issues, you must understand how SQL Server uses locks, which we describe in detail in Skill 3.1. As part of the troubleshooting process, you need to determine which resources are locked, why they are locked, and the lock type in effect.

You can use the following dynamic management views (DMVs) to view information about locks:

- **sys.dm_tran_locks** Use this DMV to view all current locks, the lock resources, lock mode, and other related information.
- **sys.dm_os_waiting_tasks** Use this DMV to see which tasks are waiting for a resource.
- **sys.dm_os_wait_stats** Use this DMV to see how often processes are waiting while locks are taken.

Before we look at these DMVs in detail, let's set up our environment as shown in Listing 3-7 so that we can establish some context for locking behavior.

LISTING 3-7 Create a test environment for testing locking behavior

Click here to view code image

```
CREATE TABLE Examples.LockingA
(
    RowId  int NOT NULL
        CONSTRAINT PKLockingARowId PRIMARY KEY,
    ColumnText  varchar(100) NOT NULL
);

INSERT INTO Examples.LockingA(RowId, ColumnText)
VALUES (1, 'Row 1'), (2, 'Row 2'), (3, 'Row 3'), (4, 'Row 4');
CREATE TABLE Examples.LockingB
(
    RowId  int NOT NULL
        CONSTRAINT PKLockingBRowId PRIMARY KEY,
    ColumnText  varchar(100) NOT NULL
);

INSERT INTO Examples.LockingB(RowId, ColumnText)
VALUES (1, 'Row 1'), (2, 'Row 2'), (3, 'Row 3'), (4, 'Row 4');
```

sys.dm_tran_locks

The sys.dm_tran_locks DMV provides you with information about existing locks and locks that have been requested but not yet granted in addition to details about the resource for which the lock is requested. You can use this DMV only to view information at the current point in time. It does not provide access to historical information about locks. Table 3-2 describes each column in sys.dm_tran_locks.

COLUMN	DESCRIPTION
resource_type	One of the following types of resources: DATABASE, FILE, OBJECT, PAGE, KEY, EXTENT, RID, APPLICATION, METADATA, HOBT, or ALLOCATION_UNIT.
resource_subtype	If a resource has a subtype, this column displays it.
resource_database_id	The ID of the database containing the resource.
resource_description	Additional information, if available, about the resource not found in other resource columns.
resource_associated_entity_id	The ID of the entity with which the resource is associated, such as an object ID, HoBT ID, or Allocation Unit ID.
resource_lock_partition	The ID of the lock partition for partitioned lock resource. The value is 0 for a non-partitioned lock resource.
request_mode	The lock mode requested by waiting requests or granted for other requests.
request_type	This value is always LOCK.
request_status	One of the following values to reflect the current status of the request: GRANTED, CONVERT, WAIT, LOW_PRIORITY_CONVERT, LOW_PRIORITY_WAIT, or ABORT_BLOCKERS.
request_reference_count	The approximate number of times the requestor has requested the resource.
request_lifetime	This column is not supported.
request_session_id	The ID of the session that owns the request. An orphaned distributed transaction has a value of -2. A deferred recovery transaction has a value of -3.
request_exec_context_id	The ID of the execution context for the process that owns the request.
request_request_id	The ID of the request for the process that owns the request. This value changes when the active MARS connection for the transaction changes.
request_owner_type	The type of entity that owns the request: TRANSACTION, CURSOR, SESSION, SHARED_TRANSACTION_WORKSPACE, EXCLUSIVE_TRANSACTION_WORKSPACE, or NOTIFICATION_OBJECT.
request_owner_id	The ID of the owner of the request.
request_owner_guid	The GUID of the owner of the request.
request_owner_lockspace_id	This column is not supported.
lock_owner_address	The memory address of the internal data structure that is tracking the request. Join it with the resource_address column in sys.dm_os_waiting_tasks.
pdw_node_id	The ID for node in the Analytics Platform System (formerly known as Parallel Data Warehouse).

TABLE 3-2 sys.dm_tran_locks

Note Sys.dm_tran_locks resource type subtypes

For a full list of subtypes for each resource type, refer to the sys.dm_tran_locks documentation at *https://msdn.microsoft.com/en-us/library/ms190345.aspx*.

Let's start some transactions to observe the locks that SQL Server acquires. In one session, execute the following statements:

Click here to view code image

```
BEGIN TRANSACTION;
    SELECT RowId, ColumnText
    FROM Examples.LockingA
    WITH (HOLDLOCK, ROWLOCK);
```

In a separate session, start another transaction:

Click here to view code image

```
BEGIN TRANSACTION;
    UPDATE Examples.LockingA
        SET ColumnText = 'Row 2 Updated'
        WHERE RowId = 2;
```

Now let's use the sys.dm_tran_locks DMV to view some details about the current locks:

Click here to view code image

```
SELECT
    request_session_id as s_id,
    resource_type,
    resource_associated_entity_id,
    request_status,
    request_mode
FROM sys.dm_tran_locks
WHERE resource_database_id = db_id('ExamBook762Ch3');
```

Although your results might vary, especially with regard to identifiers, the DMV returns results similar to the example below. Notice the wait for the exclusive lock for session 2. It must wait until session 1 releases its shared range (RangeS-S) locks that SQL Server takes due to the HOLDLOCK table hint. This table hint is equivalent to setting the isolation level to SERIALIZABLE. SQL Server also takes intent locks on the table (which appears on the OBJECT rows of the results) and the page, with session 1 taking intent shared (IS) locks and session 2 taking intent exclusive (IX) locks.

Click here to view code image

```
s_id resource_type resource_associated_entity_id
request_status   request_mode
```

```
---- ------------  ------------------------------ ------------
-   --------------
1    DATABASE       0                              GRANT
2    DATABASE       0                              GRANT
1    PAGE           72057594041729024              GRANT
2    PAGE           72057594041729024              GRANT
1    KEY            72057594041729024              GRANT
S
1    KEY            72057594041729024              GRANT
S
1    KEY            72057594041729024              GRANT
S
1    KEY            72057594041729024              GRANT
S
1    KEY            72057594041729024              GRANT
S
2    KEY            72057594041729024              WAIT
1    OBJECT         933578364                      GRANT
2    OBJECT         933578364                      GRANT
```

Connect to the ExamBook762Ch3 database containing the resource and use one of the resource_associated_entity_id values from the previous query in the WHERE clause to see which object is locked, like this:

Click here to view code image

```
SELECT
    object_name(object_id) as Resource,
    object_id,
    hobt_id
FROM sys.partitions
WHERE hobt_id=72057594041729024;
```

When you view the results of this latter query, you can see the name of the resource that is locked, like this:

Click here to view code image

```
Resource object_id  hobt_id
-------- ---------- -------------------
LockingA 933578364  72057594041729024
```

In the previous example, you can also see the object_id returned from sys.partitions corresponds to the resource_associated_entity_id associated with the OBJECT resource_type in the DMV.

When troubleshooting blocking situations, look for CONVERT in the request_status column in this DMV. This value indicates the request was granted a lock mode earlier, but now needs to upgrade to a different lock mode and is currently blocked.

sys.dm_os_waiting_tasks

Another useful DMV is sys.dm_os_waiting_tasks. Whenever a user asks you why a query is taking longer to run than usual, a review of this DMV should be one of your standard troubleshooting steps. You can find a description of each column in this DMV in Table 3-3.

COLUMN	DESCRIPTION
waiting_task_address	The address of the waiting task.
session_id	The ID of the session that owns the task.
exec_context_id	The ID of the execution context of the task.
wait_duration_ms	The total wait time for this wait type in milliseconds. This value includes signal_wait_time_ms.
wait_type	The type of wait.
resource_address	The address of the resource for which the task is waiting.
blocking_task_address	The task that is currently holding the requested resource.
blocking_session_id	The ID of the session that is blocking the request. This column is NULL if the task is not blocked, -2 if the blocking resource is owned by an orphaned transaction, -3 if the blocking resource is owned by a deferred recovery transaction, and -4 if the session ID of the blocking latch owner cannot be determined due to internal latch state transitions.
blocking_exec_context_id	The ID of the execution context of the blocking task.
resource_description	The description of the resource consumed. See https://msdn.microsoft.com/en-us/library/ms188743.aspx for more information.
pdw_node_id	The ID for node in the Analytics Platform System (formerly known as Parallel Data Warehouse).

TABLE 3-3 sys.dm_os_waiting_tasks

In particular, you can use the sys.dm_trans_locks DMV in conjunction with the sys.dm_os_waiting_tasks DMV to find blocked sessions, as shown in Listing 3-8.

LISTING 3-8 Use system DMV sys.dm_tran_locks and sys.dm_os_waiting_tasks to display blocked sessions

Click here to view code image

```
SELECT
    t1.resource_type AS res_typ,
    t1.resource_database_id AS res_dbid,
    t1.resource_associated_entity_id AS res_entid,
    t1.request_mode AS mode,
    t1.request_session_id AS s_id,
    t2.blocking_session_id AS blocking_s_id
FROM sys.dm_tran_locks as t1
INNER JOIN sys.dm_os_waiting_tasks as t2
```

```
    ON t1.lock_owner_address = t2.resource_address;
```

Whereas the earlier query showing existing locks is helpful for learning how SQL Server acquires locks, the query in Listing 3-8 returns information that is more useful on a day-to-day basis for uncovering blocking chains. In the query results shown below, you can see that session 2 is blocked by session 1.

Click here to view code image

```
res_typ  res_dbid   res_entid           mode  s_id   blocking_
-------  --------  ------------------  ----  ------  --------
---------
KEY      27        72057594041729024   X     2       1
```

Execute the following statement in both sessions to release the locks:

```
ROLLBACK TRANSACTION;
```

sys.dm_os_wait_stats

The sys.dm_os_wait_stats DMV is an aggregate view of all waits that occur when a requested resource is not available, a worker thread is idle typically due to background tasks, or an external event must complete first. Table 3-4 explains the columns in sys.dm_os_wait_stats.

COLUMN	DESCRIPTION
wait_type	The type of wait. The wait types associated with locks all begin with LCK.
waiting_tasks_count	The number of waits having this wait type. The start of a new wait increments this value.
wait_time_ms	The total wait time for this wait type in milliseconds. This value includes signal_wait_time_ms.
max_wait_time_ms	The highest wait time for this wait type in milliseconds.
signal_wait_time_ms	The amount of time in milliseconds between the time the waiting thread was signaled and the time it started running.
pdw_node_idpdw_node_id	The ID for node in the Analytics Platform System (formerly known as Parallel Data Warehouse).

TABLE 3-4 sys.dm_os_wait_stats

There are many wait types unrelated to locks, so when using the sys.dm_os_wait_stats DMV, you should apply a filter to focus on lock waits only, like this:

Click here to view code image

```
SELECT
    wait_type as wait,
    waiting_tasks_count as wt_cnt,
    wait_time_ms as wt_ms,
    max_wait_time_ms as max_wt_ms,
```

```
    signal_wait_time_ms as signal_ms
FROM sys.dm_os_wait_stats
WHERE wait_type LIKE 'LCK%'
ORDER BY wait_time_ms DESC;
```

Note Wait types

For a full list of wait types, refer to the sys.dm_os_wait_stats documentation at *https://msdn.microsoft.com/en-us/library/ms179984.aspx*.

The partial results of this query on our computer shown in the following example indicate that our SQL Server instance have the longest waits when threads are waiting for an exclusive (X) lock. On the other hand, the greatest number of waits is a result of waiting for a schema modification (SCH-M) lock. In both cases, the waits are caused because SQL Server has already granted an incompatible lock to the resource on another thread. This information is useful for identifying long-term trends, but does not show you details about the locked resources.

Click here to view code image

```
wait           wt_cnt  wt_ms     max_wt_ms  signal_ms
-------------- ------- --------- ---------- ----------
LCK_M_X        6       1170670   712261     114
LCK_M_S        28      19398     2034       43
LCK_M_SCH_M    449     92        28         46
LCK_M_SCH_S    1       72        72         0
```

Note Wait type troubleshooting library

Your SQL Server instance undoubtedly yields different results for this DMV. You can find a comprehensive library of SQL Server wait types compiled by SQLSkills available at *https://www.sqlskills.com/help/waits*. This library includes a description of wait types, general guidance for troubleshooting lock waits, and specific guidance for individual lock waits.

You can reset the cumulative values in the sys.dm_os_wait_stats DMV by executing the following statement: DBCC SQLPERF (N'sys.dm_os_wait_stats', CLEAR);. Otherwise, these values are reset each time that the SQL Server service restarts.

Exam Tip

For the exam, you should know which DMVs you can reset manually as compared to the DMVs that require a SQL Server service restart to be reset.

Identify lock escalation behaviors

Lock escalation occurs when SQL Server detects too much memory, or too many system resources are required for a query's locks. It then converts one set of locks to another set of locks applied to resources higher in the lock hierarchy. In other words, SQL Server tries to use fewer locks to cover more resources. As an example, SQL Server might choose to escalate a high number of row locks to a table lock. This capability can reduce overhead on the one hand, but can impact performance on the other hand because more data is locked. As a result, there is greater potential for blocking.

Lock escalation occurs when more than 40 percent of the available database engine memory pool is required by lock resources, or at least 5,000 locks are taken in a single T-SQL statement for a single resource. SQL Server converts an intent lock to a full lock, as long as the full lock is compatible with existing locks on the resource. It then releases system resources and locks on the lower level of the lock hierarchy. If the new lock is taken on a row or a page, SQL Server adds an intent lock on the object at the next higher level. However, if other locks prevent lock escalation, SQL Server continues attempting to perform the escalation for each new 1,250 locks it takes.

In most cases, you should let SQL Server manage the locks. If you implement a monitoring system, take note of Lock:Escalation events to establish a benchmark. When the number of Lock:Escalation events exceeds the benchmark, you can take action at the table level or at the query level.

Another option for monitoring lock escalation is to benchmark the percentage of time that intent lock waits (LCK_M_I*) occur relative to regular locks in the sys.dm_os_wait_stats DMV by using a query like this:

Click here to view code image

```
SELECT
    wait_type as wait,
    wait_time_ms as wt_ms,
    CONVERT(decimal(9,2), 100.0 * wait_time_ms /
    SUM(wait_time_ms) OVER ()) as wait_pct
FROM sys.dm_os_wait_stats
WHERE wait_type LIKE 'LCK%'
ORDER BY wait_time_ms DESC;
```

Capture and analyze deadlock graphs

Usually the process of locking and unlocking SQL Server is fast enough to allow many users to read and write data with the appearance that it occurs simultaneously. However, sometimes two sessions block each other and neither can complete, which is a situation known as *deadlocking*. Normally, the database engine terminates a thread of a deadlocked transaction with error 1205 and suggests a remedy, such as running the transaction again.

Let's deliberately create a deadlock between two transactions. Start two sessions and

add the following statements to the first session:

Click here to view code image

```
BEGIN TRANSACTION;
    UPDATE Examples.LockingA
        SET ColumnText = 'Row 1 Updated'
        WHERE RowId = 1;
    WAITFOR DELAY '00:00:05';
    UPDATE Examples.LockingB;
    SET ColumnText = 'Row 1 Updated Again'
    WHERE RowId = 1;
```

Next, in the second session, add the following statements:

Click here to view code image

```
BEGIN TRANSACTION;
    UPDATE Examples.LockingB
        SET ColumnText = 'Row 1 Updated'
        WHERE RowId = 1;
    WAITFOR DELAY '00:00:05';
    UPDATE Examples.LockingA;
    SET ColumnText = 'Row 1 Updated Again'
    WHERE RowId = 1;
```

Now execute the statements in the first session, and then, within five seconds, execute the second session's statements. Only one of the transaction completes and the other was terminated with a rollback by SQL Server as shown by the following message:

Click here to view code image

```
Msg 1205, Level 13, State 51, Line 6
Transaction (Process ID 70) was deadlocked on lock resources
with another process and
has been chosen as the deadlock victim. Rerun the transaction.
```

In this example, both transactions need the same table resources. Both transactions can successfully update a row without conflict and have an exclusive lock on the updated data. Then they each try to update data in the table that the other transaction had updated, but each transaction is blocked while waiting for the other transaction's exclusive lock to be released. Neither transaction can ever complete and release its lock, thereby causing a deadlock. When SQL Server recognizes this condition, it terminates one of the transactions and rolls it back. It usually chooses the transaction that is least expensive to rollback based on the number of transaction log records. At that point, the aborted transaction's locks are released and the remaining open transaction can continue.

Of course, deadlocks are not typically going to happen while you watch, so how can you know when and why they occur? You can use either SQL Server Profiler or Extended

Events to capture a *deadlock graph*, an XML description of a deadlock.

Exam Tip

The exam also tests your knowledge about capturing deadlocks without a graph by using Trace Flags 1204 and 1222. You can enable these trace flags by using the following syntax: DBCC TRACEON(1204,1222,-1). Whenever a deadlock occurs, the deadlock victim and the other transaction involved in the deadlock appear in the SQL Server log. See "Detecting and Ending Deadlocks" at *https://technet.microsoft.com/en-us/library/ms178104.aspx* to review this topic in more depth.

SQL Server Profiler deadlock graph

If you use SQL Server Profiler to capture a deadlock graph, you must configure the trace before deadlocks occur. Start by creating a new trace, and connect to your SQL Server instance. In the Trace Properties dialog box, select the Events Selection tab, select the Show All Events check box, expand Locks, and then select the following events:

- Deadlock graph
- Lock:Deadlock
- Lock:Deadlock Chain

On the Events Extraction Settings tab, select the Save Deadlock XML Events Separately option, navigate to a directory into which SQL Server Profiler saves deadlock graphs, and supply a name for the graph. You can choose whether to save all deadlock graphs in a single .xdl file or save multiple deadlock graphs as a separate .xdl file.

Note Viewing a deadlock graph saved as an .xdl file

Whenever you can save a deadlock graph as an .xdl file, you can later open that file in SQL Server Management Studio to view it.

Now set up the deadlock scenario again to generate the deadlock graph. In one session, add the following statements:

Click here to view code image

```
BEGIN TRANSACTION;
    UPDATE Examples.LockingA
        SET ColumnText = 'Row 2 Updated'
        WHERE RowId = 2;
    WAITFOR DELAY '00:00:05';
    UPDATE Examples.LockingB
    SET ColumnText = 'Row 2 Updated Again'
```

```
    WHERE RowId = 2;
```

Next, in the second session, add the following statements:

Click here to view code image

```
BEGIN TRANSACTION;
    UPDATE Examples.LockingB
        SET ColumnText = 'Row 2 Updated'
        WHERE RowId = 2;
    WAITFOR DELAY '00:00:05';
    UPDATE Examples.LockingA
    SET ColumnText = 'Row 2 Updated Again'
    WHERE RowId = 2;
```

When a deadlock occurs, you can see the deadlock graph as an event in SQL Server Profiler, as shown in Figure 3-1. In the deadlock graph, you see the tables and queries involved in the deadlock, which process was terminated, and which locks led to the deadlock. The ovals at each end of the deadlock graph contain information about the processes running the deadlocked queries. The terminated process displays in the graph with an x superimposed on it. Hover your mouse over the process to view the statement associated with it. The rectangles labeled Key Lock identify the database object and index associated with the locking. Lines in the deadlock graph show the relationship between processes and database objects. A request relationship displays when a process waits for a resource while an owner relationship displays when a resource waits for a process.

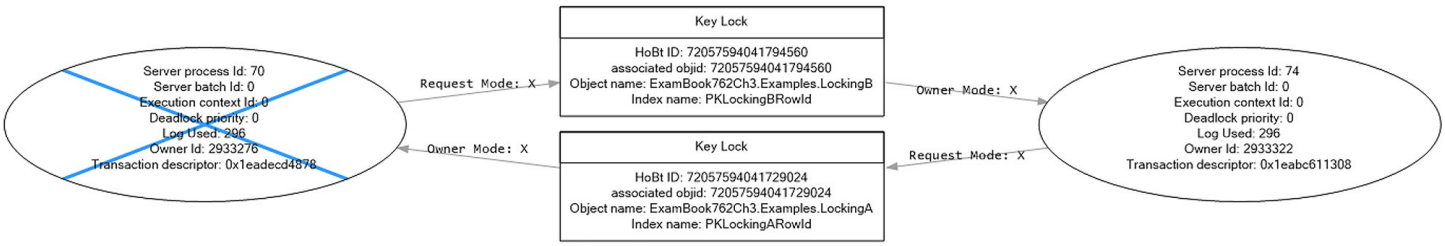


FIGURE 3-1 A deadlock graph

Extended Events deadlock graph

In Extended Events, you can use the continuously running system_health session to discover past deadlocks. As an alternative, you can set up a new session dedicated to capturing deadlock information. The system_health session automatically captures detected deadlocks without requiring special configuration. That means you can analyze a deadlock after it has occurred.

To find deadlock information in the Extended Events viewer, open SQL Server Management Studio, connect to the database engine, expand the Management node in Object Explorer, expand the Extended Events node, expand the Sessions node, and then expand the System_health node. Right-click Package0.event_file, and select View Target Data. In the Extended Events toolbar, click the Filters button. In the Filters dialog box,

select Name in the Field drop-down list, type xml_deadlock_report in the Value text box, as shown in Figure 3-2, and then click OK. Select Xml_deadlock_report in the filtered list of events, and then click the Deadlock tab below it to view the deadlock graph.

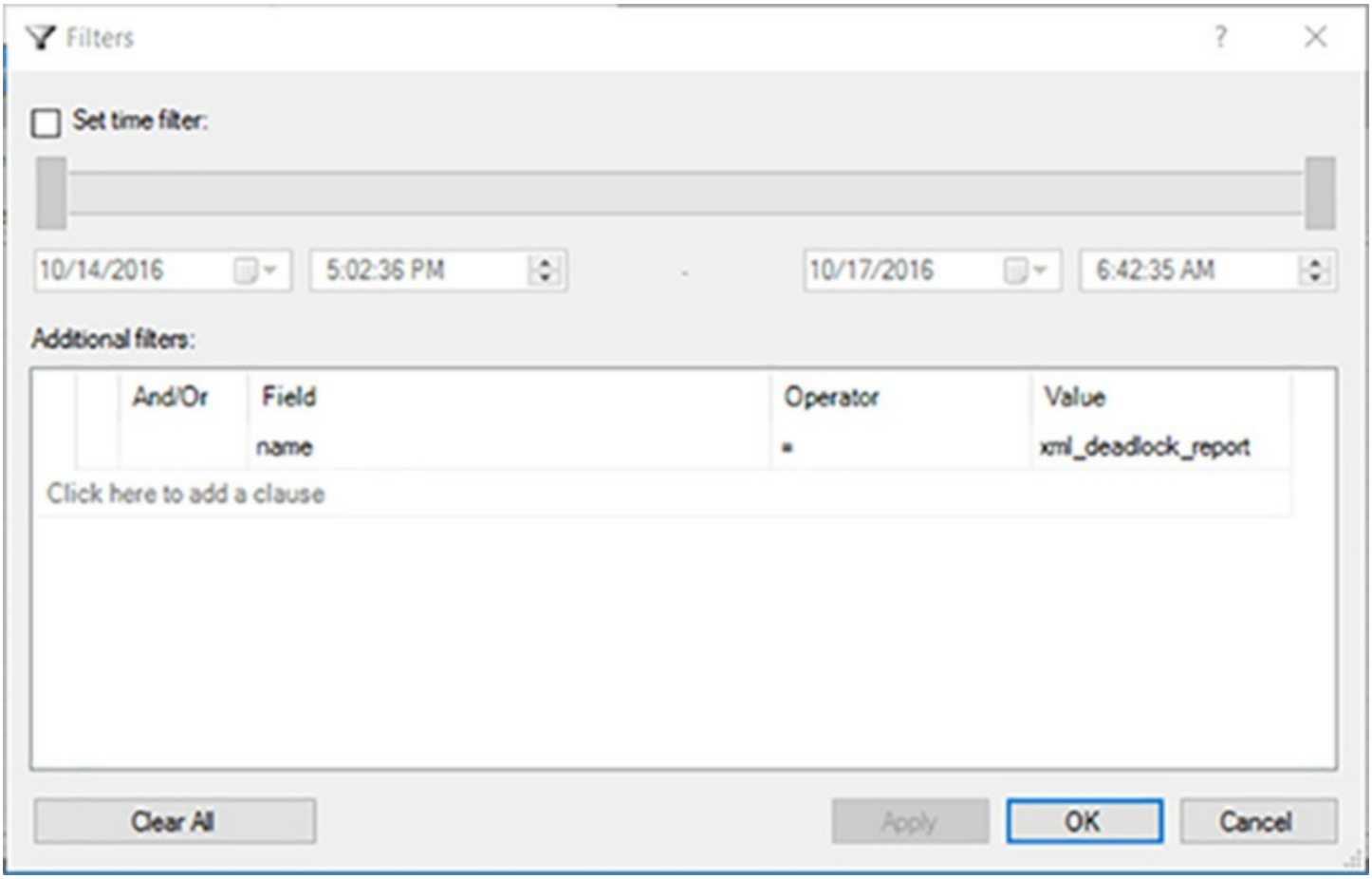


FIGURE 3-2 An Extended Events filter for xml_deadlock_report

Identify ways to remediate deadlocks

Deadlocks are less likely to occur if transactions can release resources as quickly as possible. You can also lock up additional resources to avoid contention between multiple transactions. For example, you can use a hint to lock a table although this action can also cause blocking.

Usually the best way to resolve a deadlock is to rerun the transaction. For this reason, you should enclose a transaction in a TRY/CATCH block and add retry logic. Let's revise the previous example to prevent the deadlock. Start two new sessions and add the statements in Listing 3-9 to both sessions.

LISTING 3-9 Add retry logic to avoid deadlock

Click here to view code image

```
DECLARE @Tries tinyint
SET @Tries = 1
```

```
WHILE @Tries <= 3
BEGIN

    BEGIN TRANSACTION
    BEGIN TRY
        UPDATE Examples.LockingB
            SET ColumnText = 'Row 3 Updated'
            WHERE RowId = 3;
        WAITFOR DELAY '00:00:05';
        UPDATE Examples.LockingA
        SET ColumnText = 'Row 3 Updated Again'
            WHERE RowId = 3;
        COMMIT TRANSACTION;
    END TRY
    BEGIN CATCH
        SELECT ERROR_NUMBER() AS ErrorNumber;
        ROLLBACK TRANSACTION;
        SET @Tries = @Tries + 1;
        CONTINUE;
    END CATCH
END
```

Next, execute each session. This time the deadlock occurs again, but the CATCH block captured the deadlock. SQL Server does not automatically roll back the transaction when you use this method, so you should include a ROLLBACK TRANSACTION in the CATCH block. The @@TRANCOUNT variable resets to zero in both transactions. As a result, SQL Server no longer cancels one of the transactions and you can also see the error number generated for the deadlock victim:

```
ErrorNumber
-------------
1205
```

Re-execution of the transaction might not be possible if the cause of the deadlock is still locking resources. To handle those situations, you could need to consider the following methods as alternatives for resolving deadlocks.

- Use SNAPSHOT or READ_COMMITTED_SNAPSHOT isolation levels. Either of these options avoid most blocking problems without the risk of dirty reads. However, both of these options require plenty of space in *tempdb*.
- Use the NOLOCK query hint if one of the transactions is a SELECT statement, but only use this method if the trade-off of a deadlock for dirty reads is acceptable.
- Add a new covering nonclustered index to provide another way for SQL Server to read data without requiring access to the underlying table. This approach works only if the other transaction participating in the deadlock does not use any of the covering

index keys. The trade-off is the additional overhead required to maintain the index.

- Proactively prevent a transaction from locking a resource that eventually gets locked by another transaction by using the HOLDLOCK or UPDLOCK query hints.

Skill 3.4: Implement memory-optimized tables and native stored procedures

The In-Memory OLTP feature built into SQL Server 2016 adds a new memory-optimized relational data management engine and a native stored procedure compiler to the platform that you can use to run transactional workloads with higher concurrency. A memory-optimized table is a highly optimized data structure that SQL Server uses to store data completely in memory without paging to disk. It uses hash and nonclustered ordered indexes to support faster data access from memory than traditional B-tree indexes. SQL Server maintains a copy of the data on disk to guarantee transaction durability and to reload tables during database recovery.

To further optimize query performance, you can implement natively compiled stored procedures as long as the stored procedure accesses memory-optimized tables only. A *natively compiled stored procedure* is a stored procedure compiled into machine language for faster execution, lower latency, and lower CPU utilization.

This section covers how to:

- Define use cases for memory-optimized tables
- Optimize performance of in-memory tables
- Determine best case usage scenarios for natively compiled stored procedures
- Enable collection of execution statistics for natively compiled stored procedures

Define use cases for memory-optimized tables

You use memory-optimized tables when you need to improve the performance and scalability of existing tables, or when you have frequent bottlenecks caused by locking and latching or code execution. SQL Server uses optimistic concurrency management for memory-optimized tables, which eliminates the need for locks and latches and in results in faster operations. In addition, SQL Server uses algorithms that are specifically optimized to access data from memory and natively compiled stored procedures to execute code faster. Depending on the type of workload you run, you can achieve five to 20 times performance gains with higher throughput and lower latency after migrating an existing disk-based table to a memory-optimized table.

In general, OLTP workloads with the following characteristics benefit most from

migration to memory-optimized tables: short transactions with fast response times, queries accessing a limited number of tables that contain small data sets, and high concurrency requirements. This type of workload could also require high transaction throughput and low latency at scale. In the exam, you must be able to recognize the following use cases for which memory-optimized tables are best suited:

- **High data ingestion rate** The database engine must process a high number of inserts, either as a steady stream or in bursts. Bottlenecks from locking and latching are a common problem in this scenario. Furthermore, last-page contention can occur when many threads attempt to access the same page in a standard B-tree and indexes intended to improve query performance add overhead time to insert operations. Performance is often measured in terms of throughput rate or the number of rows loaded per second. A common scenario for this workload is the Internet of Things in which multiple sensors are sending data to SQL Server. Other examples include of applications producing data at a high rate include financial trading, manufacturing, telemetry, and security monitoring. Whereas disk-based tables can have difficulty managing the rate of inserts, memory-optimized tables can eliminate resource contention and reduce logging. In some cases, requirements permitting, you can further reduce transaction execution time by implementing delayed durability, which we describe in greater detail in the next section.
- **High volume, high performance data reads** Bottlenecks from latching and locking, or from CPU utilization can occur when there are multiple concurrent read requests competing with periodic inserts and updates, particularly when small regions of data within a large table are accessed frequently. In addition, query execution time carries overhead related to parsing, optimizing, and compiling statements. Performance in this case often requires the overall time to complete a set of steps for a business transaction to be measured in milliseconds or a smaller unit of time. Industries with these requirements include retail, online banking, and online gaming, to name a few. The use of memory-optimized tables in this scenario eliminates contention between read and write operations and retrieves data with lower latency, while the use of natively compiled stored procedures enables code to execute faster.
- **Complex business logic in stored procedures** When an application requires intensive data processing routines and performs high volume inserts, updates, and deletes, the database can experience significant read-write contention. In some scenarios, the workload must process and transform data before writing it to a table, as is common in Extract-Transform-Load (ETL) operations, which can be a time-consuming operation. In other scenarios, the workload must perform point lookups or minimal joins before performing update operations on a small number of rows. Industries with these types of high-volume, complex logic workloads include manufacturing supply chains and retailers maintaining online, real-time product catalogs. Memory-optimized tables can eliminate lock and latch contention and

natively compiled stored procedures can reduce the execution time to enable higher throughput and lower latency. Another possibility is to use delayed durability to reduce transaction execution time, but only if the application requirements permit some potential data loss.

- **Real-time data access** Several factors contribute to latency when accessing data in traditional disk-based tables, including the time required to parse, optimize, compile, and execute a statement, the time to write a transaction to the transaction log, and CPU utilization when the database is experiencing heavy workloads. Examples of industries requiring low latency execution include financial trading and online gaming. With memory-optimized tables, the database engine retrieves data more efficiently with reduced contention and natively compiled stored procedures execute code more efficiently and faster. In addition, point lookup queries execute faster due to the use of non-clustered hash indexes and minimal logging reduces overall transaction execution time. Altogether, these capabilities of memory-optimized tables enable significantly lower latency than disk-based tables.
- **Session state management** Applications that require the storage of state information for stateless protocols, such as HTTP, often use a database to persist this information across multiple requests from a client. This type of workload is characterized by frequent inserts, updates, and point lookups. When running at scale with load balanced web servers, multiple servers can update data for the same session or perform point lookups, which results in increased contention for the same resources. This type of workload is characterized by frequently changes to a small amount of data and incurs a significant amount of locking and latching. Memory-optimized tables reduce contention, retrieve data effectively, and reduce or eliminate IO when using non-durable (SCHEMA_ONLY) tables, which we describe in the next section. On the other hand, the database engine does not resolve conflicts resulting from attempts by separate concurrent transactions to modify the same row, which is relatively rare in session state management. Nonetheless, the application should include logic to retry an operation if a transaction fails due to a write conflict.
- **Applications relying heavily on temporary tables, table variables, and table-valued parameters** Many times an application needs a way to store intermediate results in a temporary location before applying additional logic to transform the data into its final state for loading into a target table. Temporary tables and table variables are different ways to fulfill this need. As an alternative, an application might use table-valued parameters to send multiple rows of data to a stored procedure or function and avoid the need for temporary tables or table variables. All three of these methods require writes to tempdb and potentially incur additional execution time due to the IO overhead. You can instead use memory-optimized temporary tables, table variables, and table-valued parameters to take advantage of the same optimizations available for memory-optimized tables. By doing so, you can eliminate both the

tempdb contention and the IO overhead. You can achieve faster execution time when using these objects in a natively compiled stored procedure.

- **ETL operations** ETL operations typically require the use of staging tables to copy data from source systems as a starting point, and might also use staging tables for intermediate steps necessary to transform data prior to loading the processed data into a target table. Although this workload does not usually suffer from bottlenecks related to concurrency problems, it can experience delays due to IO operations and the overhead associated with query processing. For applications requiring low latency in the target database, consider using memory-optimized tables for faster, more efficient access to data. To reduce or eliminate IO, use non-durable tables as we describe in the next section.

Need More Review? Memory-optimized table use cases and implementation strategies

For a more in-depth review of use cases and implementation strategies for memory-optimized tables, download the "In-Memory OLTP – Common Workload Patterns and Migration Considerations" whitepaper from *https://msdn.microsoft.com/library/dn673538.aspx*. Be aware that memory-optimized tables in SQL Server 2016 support more features than described in the whitepaper, which was written about SQL Server 2014. For a complete list of the newly supported features see *https://msdn.microsoft.com/en-us/library/bb510411.aspx#InMemory*.

Optimize performance of in-memory tables

As we described in the previous section, there are many use cases for which migrating disk-based tables to memory-optimized tables improves overall performance. However, to ensure you get optimal performance, there are several tasks that you can perform.

Important SQL Server editions supporting memory-optimized tables

To use memory-optimized tables, you must use the SQL Server 2016 Enterprise, Developer, or Evaluation edition. In this latest version, the maximum size of an optimized table is 2 terabytes (TB).

Before we look at these tasks, let's start by creating the data directory on the root drive to hold a new database, and then enabling in-memory OLTP in a new database as shown in Listing 3-10. Enabling a database for memory-optimized tables requires you to define the filegroup by using the CONTAINS MEMORY_OPTIMIZED_DATA option. SQL Server uses this filegroup container to store checkpoint files necessary for recovering memory-optimized tables.

Important Manually create a data directory for the memory-optimized table examples

You must manually create the data directory on the C drive before you execute the statements shown in Listing 3-10. If the directory does not exist, the statement execution fails.

LISTING 3-10 Enable in-memory OLTP in a new database

Click here to view code image

```
CREATE DATABASE ExamBook762Ch3_IMOLTP
ON PRIMARY (
    NAME = ExamBook762Ch3_IMOLTP_data,
    FILENAME = 'C:\data\ExamBook762Ch3_IMOLTP.mdf', size=500MB
),
FILEGROUP ExamBook762Ch3_IMOLTP_FG CONTAINS
MEMORY_OPTIMIZED_DATA (
    NAME = ExamBook762Ch3_IMOLTP_FG_Container,
    FILENAME = 'C:\data\ExamBook762Ch3_IMOLTP_FG_Container'
)
LOG ON (
    NAME = ExamBook762Ch3_IMOLTP_log,
    FILENAME = 'C:\data\ExamBook762Ch3_IMOLTP_log.ldf',
size=500MB
);
GO
```

Now let's create the Examples schema, and then add one memory-optimized table and one disk-based table for comparison, as shown in Listing 3-11. Notice the addition of the MEMORY_OPTIMIZED = ON clause, which instructs the database engine to create a table dynamic link library (DLL) file and load the table into memory. The database engine also generates and compiles DML routines for accessing data in the table and saves the routines as DLLs, which are called when data manipulation is requested. Unless you specify otherwise, as we describe later in the "Durabiity options" section, a memory-optimized table is durable in which case it must have a primary key defined. Furthermore, it must also contain at least one index, which is satisfied below by the specification of NONCLUSTERED on the primary key column. We discuss indexing options in greater detail later in the "Indexes" section.

LISTING 3-11 Create a new schema and add tables to the memory-optimized database

Click here to view code image

```
USE ExamBook762Ch3_IMOLTP;
GO
CREATE SCHEMA Examples;
GO
CREATE TABLE Examples.Order_Disk (
    OrderId INT NOT NULL PRIMARY KEY NONCLUSTERED,
    OrderDate DATETIME NOT NULL,
    CustomerCode NVARCHAR(5) NOT NULL
);
GO
CREATE TABLE Examples.Order_IM (
    OrderID INT NOT NULL PRIMARY KEY NONCLUSTERED,
    OrderDate DATETIME NOT NULL,
    CustomerCode NVARCHAR(5) NOT NULL
)
WITH (MEMORY_OPTIMIZED = ON);
GO
```

Note Memory-optimized table row size considerations and unsupported data types

SQL Server 2016 supports a row size greater than 8,060 bytes, even without a LOB column, in a memory-optimized table. However, for better performance, you should create as narrow a table as possible.

You should be aware that SQL Server 2016 does not support the following data types in memory-optimized tables: datetimeoffset, geography, geometry, hierarchyid, rowversion, xml, sql_variant, and user-defined types.

Natively compiled stored procedures

The first optimization task to perform is to create a natively compile stored procedure. Natively compiled stored procedures are compiled at the time of creation, unlike interpreted stored procedures that compile at first execution. Furthermore, natively compiled stored procedures can access memory-optimized tables only. Native compilation translates the stored procedure code first into C code, and then into machine language, which enables the business logic to both execute and access data faster and more efficiently. The machine language version of the code is stored as a dynamic link library (DLL) in the default data directory for your SQL Server instance.

Many of the limitations that existed in SQL Server 2014 for natively compiled stored procedures have been removed. However, the following limitations still remain in SQL Server 2016:

- **tempdb access** You cannot create or access temporary tables, table variables, or table-valued functions in tempdb. Instead, you can create a non-durable memory-

optimized table (described later in this section) or memory-optimized table types or table variables.

Need More Review? Memory optimization of temporary tables

For more information about replacing temporary tables with memory-optimized objects, see "Faster temp table and table variable by using memory optimization" at *https://msdn.microsoft.com/en-us/library/mt718711.aspx*.

- **Cursors** As an alternative, you can use set-based logic or a WHILE loop.
- **CASE statement** To work around lack of support for the CASE statement, you can use a table variable to store the result set. The table variable includes a column to serve as a condition flag that you can then use as a filter.

Note workaround for a CASE statement in a natively compiled stored procedure

You can see an example of code implementing a workaround for the conditional logic of a `CASE` statement in "Implementing a CASE Expression in a Natively Compiled Stored Procedure" at *https://msdn.microsoft.com/en-us/library/dn629453.aspx*.

- **MERGE statement** You cannot use a memory-optimized table as the target of a MERGE statement. Therefore, you must use explicit INSERT, UPDATE, or DELETE statements instead.
- **SELECT INTO clause** You cannot use an INTO clause with a SELECT statement. As an alternative, use INSERT INTO <table> SELECT syntax.
- **FROM clause** You cannot use a FROM clause or subqueries in an UPDATE statement.

Note Workaround for a FROM clause in a natively compiled stored procedure.

As a workaround, you can use a memory-optimized type and a trigger as described in "Implementing UPDATE with FROM or Subqueries" at *https://msdn.microsoft.com/en-us/library/mt757375.aspx*.

- **PERCENT or WITH TIES in TOP clause** There are no alternatives for using these options in a natively compiled stored procedure.
- **DISTINCT with aggregate functions** There is no alternative for using this option in a natively compiled stored procedure.

- **Operators: INTERSECT, EXCEPT, APPLY, PIVOT, UNPIVOT, LIKE, CONTAINS** There are no alternatives for using these operators in a natively compiled stored procedure.
- **Common table expressions (CTEs)** You must rewrite your query to reproduce the functionality of a CTE in a natively compiled stored procedure.
- **Multi-row INSERT statements** You must instead use separate INSERT statements in a natively compiled stored procedure.
- **EXECUTE WITH RECOMPILE** There is no alternative for using this option in a natively compiled stored procedure.
- **View** You cannot reference a view in a natively compiled stored procedure. You must define your desired SELECT statement explicitly in the procedure code.

Note Unsupported T-SQL constructs for natively compiled stored procedures

The list of limitations for natively compiled stored procedures highlights the main features that remain unsupported. You can find a complete list of constructs, features, operators, and so on for natively compiled stored procedures at "Transact-SQL Constructs Not Supported by In-Memory OLTP" at *https://msdn.microsoft.com/en-us/library/dn246937.aspx*.

If you want to migrate an existing stored procedure to a natively compiled stored procedure, you can use the Stored Procedure Native Compilation Advisor in SQL Server Management Studio to evaluate whether your stored procedure contains elements that are not supported. You can learn more about this tool by reading "Native Compilation Advisor" at *https://msdn.microsoft.com/en-us/library/dn358355.aspx*.

Exam Tip

For the exam, it is important that you are able to identify the limitations of natively compiled stored procedures.

To observe the performance difference between an interpreted stored procedure and a natively compiled stored procedure, create two stored procedures as shown in Listing 3-12. In this example, the following portions of the code are specific to native compilation:

- **WITH NATIVE_COMPILATION** This clause is required to create a natively compiled stored procedure.
- **SCHEMABINDING** This option is required to bind the natively compiled stored procedure to the object that it references. Consequently, you cannot drop tables

referenced in the procedure code. Furthermore, you cannot use the wildcard (*) operator, and instead must reference column names explicitly in a SELECT statement.

- **BEGIN ATOMIC...END** You use this option to create an atomic block, which is a block of T-SQL statements that succeed or fail together. A natively compiled stored procedure can have only one atomic block. Starting an atomic block creates a transaction if one does not yet exist or creates a savepoint if there is an existing transaction. An atomic block must include options defining the isolation level and language like this: WITH (TRANSACTION ISOLATION LEVEL = SNAPSHOT, LANGUAGE = N'English').

LISTING 3-12 Create stored procedures to test execution performance

Click here to view code image

```
USE ExamBook762Ch3_IMOLTP;
GO
-- Create natively compiled stored procedure
CREATE PROCEDURE Examples.OrderInsert_NC
    @OrderID INT,
    @CustomerCode NVARCHAR(10)
WITH NATIVE_COMPILATION, SCHEMABINDING
AS
BEGIN ATOMIC
WITH (TRANSACTION ISOLATION LEVEL = SNAPSHOT, LANGUAGE =
N'English')
    DECLARE @OrderDate DATETIME = getdate();
    INSERT INTO Examples.Order_IM (OrderId, OrderDate,
CustomerCode)
    VALUES (@OrderID, @OrderDate, @CustomerCode);
END;
GO
-- Create interpreted stored procedure
CREATE PROCEDURE Examples.OrderInsert_Interpreted
    @OrderID INT,
    @CustomerCode NVARCHAR(10),
    @TargetTable NVARCHAR(20)
AS
    DECLARE @OrderDate DATETIME = getdate();
    DECLARE @SQLQuery NVARCHAR(MAX);
    SET @SQLQuery = 'INSERT INTO ' +
        @TargetTable +
        ' (OrderId, OrderDate, CustomerCode) VALUES (' +
        CAST(@OrderID AS NVARCHAR(6)) +
        ','''' +  CONVERT(NVARCHAR(20), @OrderDate, 101)+
```

```
        ''',''' +  @CustomerCode +
        ''')';
    EXEC (@SQLQuery);
GO
```

Next, run the statements at least twice in Listing 3-13 to compare the performance of each type of stored procedure. Ignore the results from the first execution because the duration is skewed due to memory allocation and other operations that SQL Server performs one time only. The code in Listing 3-13 first inserts 100,000 rows into a disk-based table using an interpreted stored procedure and measures the time required to perform the INSERT operation. Then the code inserts rows into a memory-optimized table using the same interpreted stored procedure and measures the processing time. Last, the code deletes rows from the memory-optimized table, resets the time measurement variables, and then inserts rows into the table by using a natively compiled stored procedure.

LISTING 3-13 Execute each stored procedure to compare performance

Click here to view code image

```
SET STATISTICS TIME OFF;
SET NOCOUNT ON;

DECLARE @starttime DATETIME = sysdatetime();
DECLARE @timems INT;
DECLARE @i INT = 1;
DECLARE @rowcount INT = 100000;
DECLARE @CustomerCode NVARCHAR(10);

--Reset disk-based table
TRUNCATE TABLE Examples.Order_Disk;

-- Disk-based table and interpreted stored procedure
BEGIN TRAN;
    WHILE @i <= @rowcount
    BEGIN;
        SET @CustomerCode = 'cust' + CAST(@i as NVARCHAR(6));
        EXEC Examples.OrderInsert_Interpreted @i,
@CustomerCode, 'Examples.Order_Disk';
        SET @i += 1;
    END;
COMMIT;

SET @timems = datediff(ms, @starttime, sysdatetime());
SELECT 'Disk-based table and interpreted stored procedure: ' AS
```

```
[Description],
    CAST(@timems AS NVARCHAR(10)) + ' ms' AS Duration;
-- Memory-based table and interpreted stored procedure
SET @i = 1;
SET @starttime = sysdatetime();

BEGIN TRAN;
    WHILE @i <= @rowcount
    BEGIN;
        SET @CustomerCode = 'cust' + CAST(@i AS NVARCHAR(6));
        EXEC Examples.OrderInsert_Interpreted @i,
@CustomerCode, 'Examples.Order_IM';
        SET @i += 1;
    END;
COMMIT;

SET @timems = datediff(ms, @starttime, sysdatetime());
SELECT 'Memory-optimized table and interpreted stored
procedure: ' AS [Description],
    CAST(@timems AS NVARCHAR(10)) + ' ms' AS Duration;

-- Reset memory-optimized table
DELETE FROM Examples.Order_IM;
SET @i = 1;
SET @starttime = sysdatetime();

BEGIN TRAN;
    WHILE @i <= @rowcount
    BEGIN;
        SET @CustomerCode = 'cust' + CAST(@i AS NVARCHAR(6));
        EXEC Examples.OrderInsert_NC @i, @CustomerCode;
        SET @i += 1;
    END;
COMMIT;

SET @timems = datediff(ms, @starttime, sysdatetime());
SELECT 'Memory-optimized table and natively compiled stored
procedure:'
    AS [Description],
    CAST(@timems AS NVARCHAR(10)) + ' ms' AS Duration;
GO
```

Your results vary from the results shown in the following example due to differences in hardware and memory configuration. However, your results should similarly reflect a variance in duration between the types of tables and stored procedures such that the memory-optimized table and natively compiled stored procedure performing inserts is

considerably faster than the other two options:

Click here to view code image

```
Description                                          Duration
---------------------------------------------------- ---------
Disk-based table and interpreted stored procedure:   10440 ms

Description                                                  Durat
------------------------------------------------------------ -----
----
Memory-optimized table and interpreted stored procedure:   10041
ms

Description
------------------------------------------------------------------
---------
Memory-optimized table and natively compiled stored
procedure:  1885 ms
```

> **Need More Review? Natively-compiled user-defined functions and inline table-valued functions**
>
> You can also natively compile scalar user-defined functions (UDFs) and inline table-valued functions (TVFs) in SQL Server 2016 for more efficient data access. You can learn more by reviewing "Scalar User-Defined Functions for In-Memory OLTP" at *https://msdn.microsoft.com/en-us/library/dn935012.aspx*.

Indexes

A memory-optimized table can have up to eight non-clustered indexes, all of which are covering indexes. That is, they include all columns in the table. Unlike a traditional B-tree index for a disk-based table, an index for a memory-optimized table exists only in memory and does not contain data. Instead, an index points to a row in memory and is recreated during database recovery. In addition, updates to an indexed memory-optimized table do not get logged.

An index for a memory-optimized table can be one of the following three types:

- **Hash** You use a nonclustered hash index when you have many queries that perform point lookups, also known as equi-joins. When you specify the index type, as shown below, you must include a bucket count. The bucket count value should be between one to two times the expected number of distinct values in the indexed column. It is better to have a bucket count that is too high rather than set it too low because it is more likely to retrieve data faster, although it consumes more memory.

Click here to view code image

```
CREATE TABLE Examples.Order_IM_Hash (
    OrderID INT NOT NULL PRIMARY KEY
        NONCLUSTERED HASH WITH (BUCKET_COUNT = 1000000),
    OrderDate DATETIME NOT NULL,
    CustomerCode NVARCHAR(5) NOT NULL
        INDEX ix_CustomerCode HASH WITH (BUCKET_COUNT =
1000000)
)
WITH (MEMORY_OPTIMIZED = ON);
```

- **Columnstore** A new feature in SQL Server 2016 is the ability to add a columnstore index to a memory-optimized table. This type of index, which we cover in greater detail in Chapter 1, "Design and implement database objects," is best when your queries perform large scans of a table because it can process data by using batch execution mode. Rather than read data row by row, it can process chunks of data in batches and thereby reduce query execution time and CPU utilization. Consider this type of index for single-threaded queries, sort operations (such as ORDER BY), and T-SQL window functions.

Click here to view code image

```
CREATE TABLE Examples.Order_IM_CCI (
    OrderID INT NOT NULL PRIMARY KEY NONCLUSTERED,
    OrderDate DATETIME NOT NULL,
    CustomerCode NVARCHAR(5) NOT NULL,
    INDEX ix_CustomerCode_cci CLUSTERED COLUMNSTORE)
WITH (MEMORY_OPTIMIZED = ON);
```

- **Nonclustered B-tree** You use a memory-optimized nonclustered B-tree index when your queries have an ORDER BY clause on an indexed column, or when your queries return a few records by performing range selections against an index column using the greater than (>) or less than (<) operators, or testing an indexed column for inequality. You also can consider using a nonclustered index in combination with a columnstore index when your queries perform point lookups or need to join together two fact tables in a data warehouse.

Click here to view code image

```
CREATE TABLE Examples.Order_IM_NC (
    OrderID INT NOT NULL PRIMARY KEY NONCLUSTERED,
    OrderDate DATETIME NOT NULL,
    CustomerCode NVARCHAR(5) NOT NULL INDEX ix_CustomerCode
NONCLUSTERED
)
WITH (MEMORY_OPTIMIZED = ON);
```

A new feature in SQL Server 2016 is the ability to add or drop indexes, or change the bucket count for an index in a memory-optimized table. To do this, you use the ALTER TABLE statement only, as shown in Listing 3-14. The CREATE INDEX, DROP INDEX, and ALTER INDEX statements are invalid for memory-optimized tables.

LISTING 3-14 Use the `ALTER TABLE` statement to add, modify, or drop an index

Click here to view code image

```
USE ExamBook762Ch3_IMOLTP;
GO
-- Add a column and an index
ALTER TABLE Examples.Order_IM
    ADD Quantity INT NULL,
    INDEX ix_OrderDate(OrderDate);
-- Alter an index by changing the bucket count
ALTER TABLE Examples.Order_IM_Hash
    ALTER INDEX ix_CustomerCode
        REBUILD WITH ( BUCKET_COUNT = 2000000);
-- Drop an index
ALTER TABLE Examples.Order_IM
    DROP INDEX ix_OrderDate;
```

Offlload analytics to readable secondary

The ability to use both columnstore and nonclustered indexes in memory-optimized tables makes it much easier to support both OLTP and analytics workloads in the same database. However, sometimes analytics queries require considerable CPU, IO, and memory resources that might have an adverse impact on OLTP performance. If you need to support both OLTP and analytics workloads, consider an Always On configuration to offload analytics workloads to a readable secondary.

Durability options

When you create a memory-optimized table, you must decide how SQL Server should manage durability of the data. You can choose one of the following two types:

- **Durable** With this type, SQL Server guarantees full durability just as if the table were disk-based. If you do not specify the durability option explicitly when you create a memory-optimized table, it is durable by default. To explicitly define a durable table, use the SCHEMA_AND_DATA durability option like this:

Click here to view code image

```
CREATE TABLE Examples.Order_IM_Durable (
    OrderID INT NOT NULL PRIMARY KEY NONCLUSTERED,
```

```
    OrderDate DATETIME NOT NULL,
    CustomerCode NVARCHAR(5) NOT NULL
)
WITH (MEMORY_OPTIMIZED = ON, DURABILITY=SCHEMA_AND_DATA);
GO
```

- **Non-durable** By choosing this type of durability, you instruct SQL Server to persist only the table schema, but not the data. This option is most appropriate for use cases in which data is transient, such as an application's session state management, or ETL staging. SQL Server never writes a non-durable table's data changes to the transaction log. To define a non-durable table, use the SCHEMA_ONLY durability option like this:

Click here to view code image

```
CREATE TABLE Examples.Order_IM_Nondurable (
    OrderID INT NOT NULL PRIMARY KEY NONCLUSTERED,
    OrderDate DATETIME NOT NULL,
    CustomerCode NVARCHAR(5) NOT NULL
)
WITH (MEMORY_OPTIMIZED = ON, DURABILITY=SCHEMA_ONLY);
GO
```

Because non-durable memory-optimized tables do not incur logging overhead, transactions writing to them run faster than write operations on durable tables. However, to optimize performance of durable memory-optimized tables, configure delayed durability at the database or transaction level. Just as with disk-based tables, delayed durability for a memory-optimized table reduces the frequency with which SQL Server flushes log records to disk and enables SQL Server to commit transactions before writing log records to disk.

> **Note Delayed durability usage**
>
> Use delayed durability with care. You can lose some transactions if a system failure occurs.

If you set delayed durability at the database level, every transaction that commits on the database is delayed durable by default, although you can override this behavior at the transaction level. Similarly, if the database is durable, you can configure the database to allow delayed durable transactions and then explicit define a transaction as delayed durable. If you prefer, you can disable delayed durability and prevent delayed durable transactions entirely regardless of the transaction's commit level. You can also specify delayed durability for a natively compiled stored procedure. Listing 3-15 includes examples of these various settings.

LISTING 3-15 Configure delayed durability

Click here to view code image

```
--Set at database level only, all transactions commit as
delayed durable
ALTER DATABASE ExamBook762Ch3_IMOLTP
    SET DELAYED_DURABILITY = FORCED;
--Override database delayed durability at commit for durable
transaction
BEGIN TRANSACTION;
    INSERT INTO Examples.Order_IM_Hash
    (OrderId, OrderDate, CustomerCode)
    VALUES (1, getdate(), 'cust1');
COMMIT TRANSACTION WITH (DELAYED_DURABILITY = OFF);
GO

--Set at transaction level only
ALTER DATABASE ExamBook762Ch3_IMOLTP
    SET DELAYED_DURABILITY = ALLOWED;
BEGIN TRANSACTION;
    INSERT INTO Examples.Order_IM_Hash
    (OrderId, OrderDate, CustomerCode)
    VALUES (2, getdate(), 'cust2');
COMMIT TRANSACTION WITH (DELAYED_DURABILITY = ON);

--Set within a natively compiled stored procedure
CREATE PROCEDURE Examples.OrderInsert_NC_DD
    @OrderID INT,
    @CustomerCode NVARCHAR(10)
WITH NATIVE_COMPILATION, SCHEMABINDING
AS
BEGIN ATOMIC
WITH (DELAYED_DURABILITY = ON,
        TRANSACTION ISOLATION LEVEL = SNAPSHOT, LANGUAGE =
N'English')
    DECLARE @OrderDate DATETIME = getdate();
    INSERT INTO Examples.Order_IM (OrderId, OrderDate,
CustomerCode)
    VALUES (@OrderID, @OrderDate, @CustomerCode);
END;
GO
--Disable delayed durability completely for all transactions
--    and natively compiled stored procedures
ALTER DATABASE ExamBook762Ch3_IMOLTP
    SET DELAYED_DURABILITY = DISABLED;
```

Determine best case usage scenarios for natively compiled stored procedures

In SQL Server 2016, you can use natively compiled stored procedures to get better performance when operating on memory-optimized tables. You use them for:

- Applications for which obtaining the best possible performance is a requirement
- Queries that execute frequently
- Tasks that must perform as fast as possible

If you have a lot of rows to process and a lot of logic to apply, the natively compiled stored procedure performs faster than an interpreted stored procedure. It is also good when you need to perform any of the following tasks:

- Aggregation
- Nested loop join
- Multi-statement SELECT, INSERT, UPDATE, or DELETE operations
- Complex expressions
- Procedural logic, such as conditional statements and loops

It is not typically the best option when you need to process only a single row.

Enable collection of execution statistics for natively compiled stored procedures

The goal of using memory-optimized tables is to execute processes as quickly as possible. Consequently, you could be surprised that some statistics, such as worker_time and elapsed_time, do not get collected by DMVs such as sys.dm_exec_query_stats and sys.dm_exec_procedure_stats. In fact, these DMVs include no information about natively compiled stored procedures.

Instead, you need to specifically enable the collection of execution statistics by using one of the following system stored procedures:

- **sys.sp_xtp_control_proc_exec_stats** Use this system stored procedure to enable statistics collection for your SQL Server instance at the procedure level.
- **sys.sp_xtp_control_query_exec_stats** Use this system stored procedure to enable statistics collection at the query level for selected natively compiled stored procedures.

> **Note Execution statistics collection**
>
> Keep in mind that enabling the collection of execution statistics can have an adverse effect on the performance of natively compiled stored procedures. Rather than collect statistics globally for an instance, you should collect statistics for selected natively compiled stored procedures only to reduce this impact.

sys.sp_xtp_control_proc_exec_stats

Use the sys.sp_xtp_control_proc_exec_stats system stored procedure to enable and disable procedure-level statistics collection on your SQL Server instance, as shown in Listing 3-16. When SQL Server or a database starts, statistics collection is automatically disabled. Note that you must be a member of the sysadmin role to execute this stored procedure.

LISTING 3-16 Enable and disable statistics collection at the procedure level

Click here to view code image

```
--Enable statistics collection at the procedure level
EXEC sys.sp_xtp_control_proc_exec_stats @new_collection_value =
1;

--Check the current status of procedure-level statistics
collection
DECLARE @c BIT;
EXEC sys.sp_xtp_control_proc_exec_stats
@old_collection_value=@c output
SELECT @c AS 'Current collection status';

--Disable statistics collection at the procedure level
EXEC sys.sp_xtp_control_proc_exec_stats @new_collection_value =
0;
```

sys.sp_xtp_control_query_exec_stats

Listing 3-17 shows an example of using the sys.sp_xtp_control_query_exec_stats system procedure to enable and disable query-level statistics collection. You can even use it to enable statistics collection for a specific natively compiled stored procedure, but it must have been executed at least once before you enable statistics collection. When SQL Server starts, query-level statistics collection is automatically disabled. Note that disabling statistics collection at the procedure level does not disable any statistics collection that you have configured at the query level. As with the previous system stored procedure, you must be a member of the sysadmin role to execute sys.sp_xtp_control_query_exec_stats.

LISTING 3-17 Enable and disable statistics collection at the query level

Click here to view code image

```
--Enable statistics collection at the query level
EXEC sys.sp_xtp_control_query_exec_stats @new_collection_value
= 1;
```

```
--Check the current status of query-level statistics collection
DECLARE @c BIT;
EXEC sys.sp_xtp_control_query_exec_stats
@old_collection_value=@c output;
SELECT @c AS 'Current collection status';

--Disable statistics collection at the query level
EXEC sys.sp_xtp_control_query_exec_stats @new_collection_value
= 0;

--Enable statistics collection at the query level for a
specific
--natively compiled stored procedure
DECLARE @ncspid int;
DECLARE @dbid int;
SET @ncspid = OBJECT_ID(N'Examples.OrderInsert_NC');
SET @dbid = DB_ID(N'ExamBook762Ch3_IMOLTP')
EXEC [sys].[sp_xtp_control_query_exec_stats]
@new_collection_value = 1,
    @database_id = @dbid, @xtp_object_id = @ncspid;

--Check the current status of query-level statistics collection
for a specific
--natively compiled stored procedure
DECLARE @c bit;
DECLARE @ncspid int;
DECLARE @dbid int;
SET @ncspid = OBJECT_ID(N'Examples.OrderInsert_NC');
SET @dbid = DB_ID(N'ExamBook762Ch3_IMOLTP')
EXEC sp_xtp_control_query_exec_stats @database_id = @dbid,
    @xtp_object_id = @ncspid, @old_collection_value=@c output;
SELECT @c AS 'Current collection status';

--Disable statistics collection at the query level for a
specific
--natively compiled stored procedure
DECLARE @ncspid int;
DECLARE @dbid int;
EXEC sys.sp_xtp_control_query_exec_stats @new_collection_value
= 0,
    @database_id = @dbid, @xtp_object_id = @ncspid;
```

Statistics collection queries

After enabling statistics collections at the procedure level, you can query the sys.dm_exec_procedure_stats DMV to review the results. Listing 3-19 illustrates an example query that filters for natively compiled stored procedures. This query returns

results for the time during which statistics collection was enabled and remains available after you disable statistics collection at the procedure level.

> **Important Run Natively Compiled Stored Procedures Before Getting Procedure-Level Statistics**
>
> Be sure to execute the statements in Listing 3-13 after enabling statistics collection. Otherwise, the statement in Listing 3-18 will not return results.

LISTING 3-18 Get procedure-level statistics

Click here to view code image

```
SELECT
    OBJECT_NAME(PS.object_id) AS obj_name,
    cached_time as cached_tm,
    last_execution_time as last_exec_tm,
    execution_count as ex_cnt,
    total_worker_time as wrkr_tm,
    total_elapsed_time as elpsd_tm
FROM sys.dm_exec_procedure_stats PS
INNER JOIN sys.all_sql_modules SM
    ON SM.object_id = PS.object_id
WHERE SM.uses_native_compilation = 1;
```

Here is an example of the results from the query in Listing 3-18:

Click here to view code image

```
obj_name        cached_tm                last_exec_tm
wrkr_tm  elpsd_tm
---------       ----------------------- -----------------------
------ -------- --------
OrderInsert_NC 2016-10-15 20:44:33.917 2016-10-15 20:44:35.273
100000 376987   383365
```

You can also review the statistics collection at the query level by executing a query against the sys.dm_exec_query_stats DMV, as shown in Listing 3-19.

> **Important Run Natively Compiled Stored Procedures Before Getting Query-Level Statistics**
>
> You must execute the statements in Listing 3-13 after enabling statistics collection to see results from executing the statement in Listing 3-19.

LISTING 3-19 Get query-level statistics

Click here to view code image

```
SELECT
    st.objectid as obj_id,
    OBJECT_NAME(st.objectid) AS obj_nm,
    SUBSTRING(st.text,
        (QS.statement_start_offset / 2 ) + 1,
        ((QS.statement_end_offset - QS.statement_start_offset)
/ 2) + 1)
            AS 'Query',
    QS.last_execution_time as last_exec_tm,
    QS.execution_count as ex_cnt
FROM sys.dm_exec_query_stats QS
CROSS APPLY sys.dm_exec_sql_text(sql_handle) st
INNER JOIN sys.all_sql_modules SM
    ON SM.object_id = st.objectid
WHERE SM.uses_native_compilation = 1
```

The information available in the query results from Listing 3-19 is similar to the procedure-level statistics, but includes a row for each statement in the natively compiled stored procedure and includes the query text for each statement. Note that total_worker_time and total_elapsed_time were excluded from this example to restrict the width of the query results.

Click here to view code image

```
obj_id      obj_name     Query                    last_exec_tm
-------   -------------- ---------------------- ---------------
-------- -------
981578535 OrderInsert_NC INSERT INTO            2016-10-15
21:09:25.877 100000
                          Examples.Order_IM
                          (OrderId, OrderDate,
                          CustomerCode)
                          VALUES (@OrderID,
                          @OrderDate,
                          @CustomerCode)
```

Chapter summary

- Transaction management is the key to the SQL Server support of ACID. ACID properties determine whether a set of statements are handled individually or as an indivisible unit of work, whether a transaction violates database rules, whether one transaction can see the effects of other transactions, and whether a statement persists

after an unexpected shutdown.

- SQL Server guarantees ACID by managing the effects of a transaction's success or failure through committing or rolling back a transaction, using a default isolation to prevent changes made by one transaction from impacting other transactions, and relying on a transaction log for durability.
- Implicit transactions start automatically for specific DML statements, but require an explicit COMMIT TRANSACTION or ROLLBACK TRANSACTION statement to end. Before using implicit transactions, you must enable the implicit transaction mode.
- Explicit transactions require a BEGIN TRANSACTION statement to start and a COMMIT TRANSACTION or ROLLBACK TRANSACTION to end. You should incorporate error handling and include logic to avoid nesting transactions for more complete control over transaction behavior.
- Savepoints allow you to partially rollback a transaction to a named location. Neither the SAVE TRANSACTION nor the ROLLBACK TRANSACTION statements have an effect on the @@TRANCOUNT variable (as long as the transaction rolls back to a specific savepoint rather than completely).
- A high concurrency database can suffer from data integrity issues when a process attempts to modify data while other simultaneous processes are trying to read or modify the data. Potential side effects include dirty reads, non-repeatable reads, phantom reads, and lost updates.
- SQL Server uses resource locks to enable high-concurrency while maintaining ACID properties for a transaction. SQL Server uses a lock hierarchy on resources to protect transactions and the types of locks that SQL Server can acquire on resources. SQL Server's response to a request for a new lock when a lock already exists depends on the compatibility between the requested and existing lock modes.
- SQL Server uses isolation levels to control the degree to which one transaction has visibility into the changes made by other transactions. Each of the following isolation levels has potential side effects on data integrity and on concurrency: READ COMMITTED, READ UNCOMMITTED, REPEATABLE READ, SERIALIZABLE, SNAPSHOT, and READ_COMMITTED_SNAPSHOT.
- You can change the isolation level at the session level by using the SET TRANSACTION ISOLATION LEVEL statement or at statement level by using a table hint to raise concurrency at the risk of introducing potential side effects.
- Because SQL Server acquires different types of locks for each isolation level, raising or lowering isolation levels have varying effects on transaction performance.
- The SNAPSHOT and READ_COMMITTED_SNAPSHOT isolation levels both create copies of data and require more CPU and memory than other isolation levels. In addition, they both require adequate space in tempdb, although of the two isolation

levels, READ_COMMITTED_SNAPSHOT requires less space.

- Use the system DMVs sys.dm_tran_locks and sys.dm_os_wait_stats to find locked resources, understand why they are locked, and identify the lock mode acquired for the locked resources.
- SQL Server uses lock escalation to more effectively manage locks, but as a result can result in more blocking of transactions. Use the sys.dm_os_wait_stats DMV to monitor lock escalation events and look for ways to tune queries if performance begins to degrade due to more blocking issues.
- A deadlock graph provides you with insight into the objects involved in a deadlock and identifies the terminated process. You can capture a deadlock graph by using either SQL Server Profiler to later review deadlock events that have yet to occur or by using Extended Events to review deadlock events that have already occurred.
- Enclosing a transaction in a TRY/CATCH block to retry it is usually the best way to resolve a deadlock. Alternative methods have varying trade-offs and include using the SNAPSHOT or READ_COMMITTED_SNAPSHOT isolation levels, using the NOLOCK, HOLDLOCK, or UPDLOCK query hints, or adding a new covering nonclustered index.
- Memory-optimized tables are well-suited for specific OLTP scenarios: high data ingestion rate; high volume, high performance data reads; complex business logic in stored procedures; real-time data access; session state management; applications relying heavily on temporary tables, table variables, and table-valued parameters; and ETL operations.
- Besides implementing memory-optimized tables to improve an application's performance, you can also consider the following techniques to optimize performance even more: natively compiled stored procedures, the addition of indexes to the memory-optimized tables, the use of a readable secondary in an Always On configuration to which you can offload analytics workloads, non-durable tables, or delayed durability for transactions.
- Natively compiled stored procedures typically execute faster and are best suited for applications requiring high performance, queries that execute frequently, and tasks that must perform extremely fast. You experience better performance gains over an interpreted stored procedure when a natively compiled stored procedure must process many rows of data and apply complex logic.
- Use the system stored procedures sys.sp_xtp_control_proc_exec_stats and sys.sp_xtp_control_query_exec_stats to enable or disable the collection of execution statistics for natively compiled stored procedures at the procedure level or query level, respectively. After enabling statistics collection, use the sys.dm_exec_procedure_stats and sys.dm_exec_query_stats DMVs to review the statistics.

Thought experiment

In this thought experiment, demonstrate your skills and knowledge of the topics covered in this chapter. You can find answers to this thought experiment in the next section.

You are a database administrator at Coho Winery. Your manager has asked you to troubleshoot and resolve a number of concurrency problems in the OLTP system running on SQL Server 2016. Your manager has presented you with the following issues that users of the system are experiencing:

1. Two users ran the same report within seconds of one another. When they meet to review the results, they notice that the totals in the reports do not match. One report has more detail rows than the other report. You examine the stored procedure code that produces the report.

Click here to view code image

```
SELECT
    so.OrderID,
    OrderDate,
    ExpectedDeliveryDate,
    CustomerID,
    CustomerPurchaseOrderNumber,
    StockItemID,
    Quantity,
    UnitPrice
FROM Sales.Orders so
    WITH (NOLOCK)
INNER JOIN Sales.OrderLines sol
    WITH (NOLOCK)
    ON so.OrderID = sol.OrderID
```

What step do you recommend to ensure greater consistency in the report and what are the ramifications of making this change?

2. Users are reporting a process to update the order system is running slowly right now. Which DMVs do you use to identify the blocking process and why?
3. A new application developer is asking for help diagnosing transaction behavior. The transaction in the following code never gets committed:

Click here to view code image

```
BEGIN TRANSACTION;
    UPDATE <do something>;
    BEGIN TRANSACTION;
        UPDATE <DO SOMETHING>;
        BEGIN TRANSACTION;
            UPDATE <DO SOMETHING>;
COMMIT TRANSACTION;
```

What recommendation can you give the developer to achieve the desired result and commit all update operations?

4. An internal application captures performance and logging data from thousands of devices through a web API. Seasonally, the incoming rate of data shifts from 3,000 transactions/sec to 30,000 transactions/sec and overwhelms the database. What implementation strategy do you recommend?
5. Which indexing strategy should be used for a memory-optimized table for which the common query pattern is shown below?

Click here to view code image

```
SELECT CustomerName FROM Customer
WHERE StartDate > DateAdd(day, -7, GetUtcDate());
```

Thought experiment answers

This section contains the solutions to the thought experiment. Each answer explains the resolution to each of the issues identified in the OLTP system.

1. The use of the `NOLOCK` table hint is common in reporting applications against OLTP systems in which lack of consistency is a trade-off for faster query performance. However, when users are dissatisfied with inconsistent results, you can recommend removing this table hint and allow the default isolation in SQL Server to manage transaction isolation. Long write operations can block the report from executing. Similarly, if the report takes a long time to execute, the read operation can block write operations.
2. Start with a query that returns sys.dm_os_waiting_tasks where blocking_session_id <> 0 and session_id equals the ID for the user's session to see if anything is blocking the user's request. The following columns will give you details about the blocking situation: blocking_session_id, wait_type, and wait_duration_ms. You can join this information to sys.dm_tran_locks to discover the current locks involved by including the request_mode and resource_type columns. The request_status column provides information about the locks. A value of CONVERT in this column is an indicator that a request is blocked. You can also use the value in the resource_associated_entity_id column to find the associated object's name in sys.partitions.
3. In explicit transaction mode with nested transactions, each BEGIN TRANSACTION must correspond to a COMMIT TRANSACTION. As each new transaction starts with BEGIN TRANSACTION, the @@TRANCOUNT variable increments by 1 and each COMMIT TRANSACTION decrements it by 1. The complete transaction does not get written to disk and committed completely until @@TRANCOUNT is 0.

   While this solution is correct, a better solution is not to use nested transactions.
4. For this type of scenario, you recommend migrating the application to memory-optimized tables. The use of memory-optimized tables is well-suited for the ingestion

of high-volume inserts because it prevents the bottlenecks commonly resulting from locking and eliminates logging. Consequently, the throughput rate (number of rows loaded per second) can substantially increase.

5. You should recommend a nonclustered B-tree index for this query pattern. It works best for range selections in contrast to a hash index which works best for point lookups or a columnstore index which works best for large table scans.

Chapter 4. Optimize database objects and SQL infrastructure

In the previous chapter, we considered how to optimize database performance by managing concurrency. Database optimization also requires you not only to understand how SQL Server runs queries efficiently and uses its resources effectively, but also how to recognize problems that prevent it from doing so. Given this, you must know how to use the tools that help you find those problems and the steps that you can take to tune the SQL Server infrastructure for better performance.

Often the first step you take to improve your SQL Server query performance is to add indexes to tables, as we described in Chapter 1, "Design and implement database objects." In Skill 4.1, we review the tasks you should perform periodically to ensure that the indexes you have are in fact helpful, and also to identify and resolve index issues. We then delve deeper into query performance optimization by exploring how to capture and analyze query plans in Skill 4.2. Then we shift our attention to the SQL Server infrastructure in Skill 4.3 to consider its impact on performance and understand how to use built-in tools to manage and troubleshoot the database engine's use of resources. We close the chapter with Skill 4.4 in which we review the tools at your disposal for ongoing monitoring of your database infrastructure so that you are able to recognize when performance begins to deviate from normal and can take proactive action to keep SQL Server running at its best.

Skills in this chapter:

- Optimize statistics and indexes
- Analyze and troubleshoot query plans
- Manage performance for database instances
- Monitor and trace SQL Server baseline performance metrics

Skill 4.1: Optimize statistics and indexes

One way to significantly improve the performance of queries is to add one or more indexes to a table. When you create an index, SQL Server creates s*tatistics*, a database object in which statistical information about the distribution of values in a column or index is stored. An index improves query performance only if it is up-to-date and selected by SQL Server to resolve queries. Therefore, you should periodically review and optimize indexes as part of your regular routine.

This section covers how to:

- Determine the accuracy of statistics and associated impact to query plans and performance

- Design statistics maintenance tasks
- Use dynamic management objects to review current index usage and identify missing indexes
- Consolidate overlapping indexes

Determine the accuracy of statistics and the associated impact to query plans and performance

SQL Server uses statistics to determine a column's *cardinality*, which is the number of rows containing a specific value in a column. For example, when a value is unique in a column, such as a primary key, the cardinality is 1. When a column is highly unique like this, it is commonly characterized as having high selectivity or high cardinality. Conversely, when the number of unique values in a column are fewer or non-existent, the column has low selectivity and low cardinality. This information influences the query optimizer's selection of an appropriate index as well as an efficient query plan. If the wrong index or query plan is selected, or if an index is missing, a query might not execute as optimally as possible. Therefore, the accuracy of the statistics is critical to query performance.

There are several ways that you can manage statistics. First, you can review the current statistics to determine whether the distribution of the data and the cardinality of data accessed by an index is likely to be helpful. In addition, you can control whether SQL Server updates statistics automatically. Last, you can check which objects have statistics managed by SQL Server and when they were last updated.

Review data distribution and cardinality

To see statistics for a table, use the DBCC SHOW_STATISTICS command to return the following information:

- Metadata about the statistics including date and time of the last update of the statistics, number of rows in the table, number of rows sampled, number of steps in the histogram, index density, average key length, whether the index contains string summary statistics, filtered predicate for statistics if applicable, and number of rows before applying the filter.
- Index densities for the combination of columns in the index.
- A histogram of up to 200 sample values in the first key column in the index.

Let's create a simple index for the Purchasing.Suppliers table in the WideWorldImporters database and view its statistics, as shown in Listing 4-1, to explore the output of the DBCC SHOW_STATISTICS command.

LISTING 4-1 Create an index and show its statistics

Click here to view code image

```
USE WideWorldImporters;
GO
CREATE NONCLUSTERED INDEX
IX_Purchasing_Suppliers_ExamBook762Ch4
    ON Purchasing.Suppliers
(
    SupplierCategoryID,
    SupplierID
)
INCLUDE (SupplierName);
GO

DBCC SHOW_STATISTICS ('Purchasing.Suppliers',
    IX_Purchasing_Suppliers_ExamBook762Ch4 );

/* Output
Name           Updated       Rows      Rows
Sampled      Steps   Density
Average key length   String Index     Filter Expression
Unfiltered Rows
------------------------------ -------------------- ---- ----
---------
IX_Purchasing_Suppliers_ExamBook762Ch4 Nov  7 2016  6:40AM
13  13  5    1
8                 NO                          NULL
(1 row(s) affected)

All density       Average Length Columns
----------------  ------------ ------------------------------
-----------
0.125               4                                      Supplie
0.07692308    8                           SupplierCategoryID,
SupplierID
(2 row(s) affected)

RANGE_HI_KEY  RANGE_ROWS  EQ_ROWS    DISTINCT_RANGE_ROWS  AVG_R
--------------------- -------------------- ---------------- ---
------------
2                 0                    6                    0
4                 1                    1                    1
6                 1                    1                    1
8                 1                    1                    1
9                 0                    1                    0
(5 row(s) affected)
DBCC execution completed. If DBCC printed error messages,
```

```
contact your system
administrator.
 */
```

The first result set is the statistics header that shows the name of the object for which statistics exist. In this case, the object is the IX_Purchasing_Suppliers_ExamBook762Ch4 index. You can see when the statistics were last updated, the number of rows in the index, the number of rows sampled for the statistics calculations, and the number of rows in the table without a filter. You also see a value for density in the statistics header. However, this density calculation is no longer used by the query optimizer in SQL Server 2016. Last, the average key length is 8 bytes, there are no string summary statistics, and no filter applied.

The second result set shows the densities of each combination of columns in the index. Density is calculated by dividing one by the count of distinct values in the column (or columns when you have a compound key). In Listing 4-1, the first row with SupplierCategoryID has higher density than the second row with SupplierCategoryID, SupplierID which means it has lower selectivity and is less helpful as an index. Conversely, the lower density of the second row indicates higher selectivity.

The last result set is the statistics histogram, which contains up to 200 sample values for the first column of the index. Each sample value is called a step and is listed in the RANGE_HI_KEY column. In Listing 4-1, there are only 5 sample values: 2, 4, 6, 8, and 9. For each step, SQL Server stores the following four values:

- **RANGE_ROWS** The number of rows inside the range between the current step and the previous step, but does not include the step values themselves.
- **EQ_ROWS** The number of rows having the same value as the sample value.
- **DISTINCT_RANGE_ROWS** The number of distinct values between the current step and the previous step, but does not include the step values themselves.
- **AVG_RANGE_ROWS** The average number of rows for each distinct value with the step range.

Note DBCC SHOW_STATISTICS

For more details about the DBCC SHOW_STATISTICS output, see *https://msdn.microsoft.com/en-us/library/ms174384.aspx*.

When creating an estimated query plan (described in more detail in Skill 4.2), SQL Server looks at the histogram to estimate the number of rows that match a WHERE clause in a query as long as the condition in the clause is a single constant expression, such as WHERE SupplierCategoryID = 3. When the expression uses two columns, as in WHERE SupplierCategoryID = 3 AND SupplierID = 10, SQL Server uses the index densities to

estimate rows.

Let's look at an example of how inaccurate statistics can affect a query plan. First, create a test database with automatic statistics updates disabled, load the table, add an index, and then review the statistics, as shown in Listing 4-2.

LISTING 4-2 Create test environment with automatic statistics disabled

Click here to view code image

```
CREATE DATABASE ExamBook762Ch4_Statistics;
GO
ALTER DATABASE ExamBook762Ch4_Statistics
    SET   AUTO_CREATE_STATISTICS OFF;
ALTER DATABASE ExamBook762Ch4_Statistics
    SET AUTO_UPDATE_STATISTICS OFF;
ALTER DATABASE ExamBook762Ch4_Statistics
    SET AUTO_UPDATE_STATISTICS_ASYNC OFF;
GO
USE ExamBook762Ch4_Statistics;
GO
CREATE SCHEMA Examples;
GO
CREATE TABLE Examples.OrderLines (
    OrderLineID int NOT NULL,
    OrderID int NOT NULL,
    StockItemID int NOT NULL,
    Description nvarchar(100) NOT NULL,
    PackageTypeID int NOT NULL,
    Quantity int NOT NULL,
    UnitPrice decimal(18, 2) NULL,
    TaxRate decimal(18, 3) NOT NULL,
    PickedQuantity int NOT NULL,
    PickingCompletedWhen datetime2(7) NULL,
    LastEditedBy int NOT NULL,
    LastEditedWhen datetime2(7) NOT NULL);
GO
INSERT INTO Examples.OrderLines
SELECT *
FROM WideWorldImporters.Sales.OrderLines;
GO
CREATE INDEX ix_OrderLines_StockItemID
    ON Examples.OrderLines (StockItemID);
GO
DBCC SHOW_STATISTICS ('Examples.OrderLines',
    ix_OrderLines_StockItemID );
GO
```

```
/* Partial Output
RANGE_HI_KEY   RANGE_ROWS   EQ_ROWS   DISTINCT_RANGE_ROWS   AVG_RAN
------------------ --------------- ----------- ----------------
-------- ---
1                  0                 1048                  0
2                  0                 1078                  0
4               1022                 1066                  1
*/
```

When the index is added to the table, its statistics are also created. However, a significant number of inserts or updates to the table can render these statistics obsolete. Execute the statements in Listing 4-3 to update rows and check the statistics afterwards to confirm there has been no change.

LISTING 4-3 Update table rows and check statistics

Click here to view code image

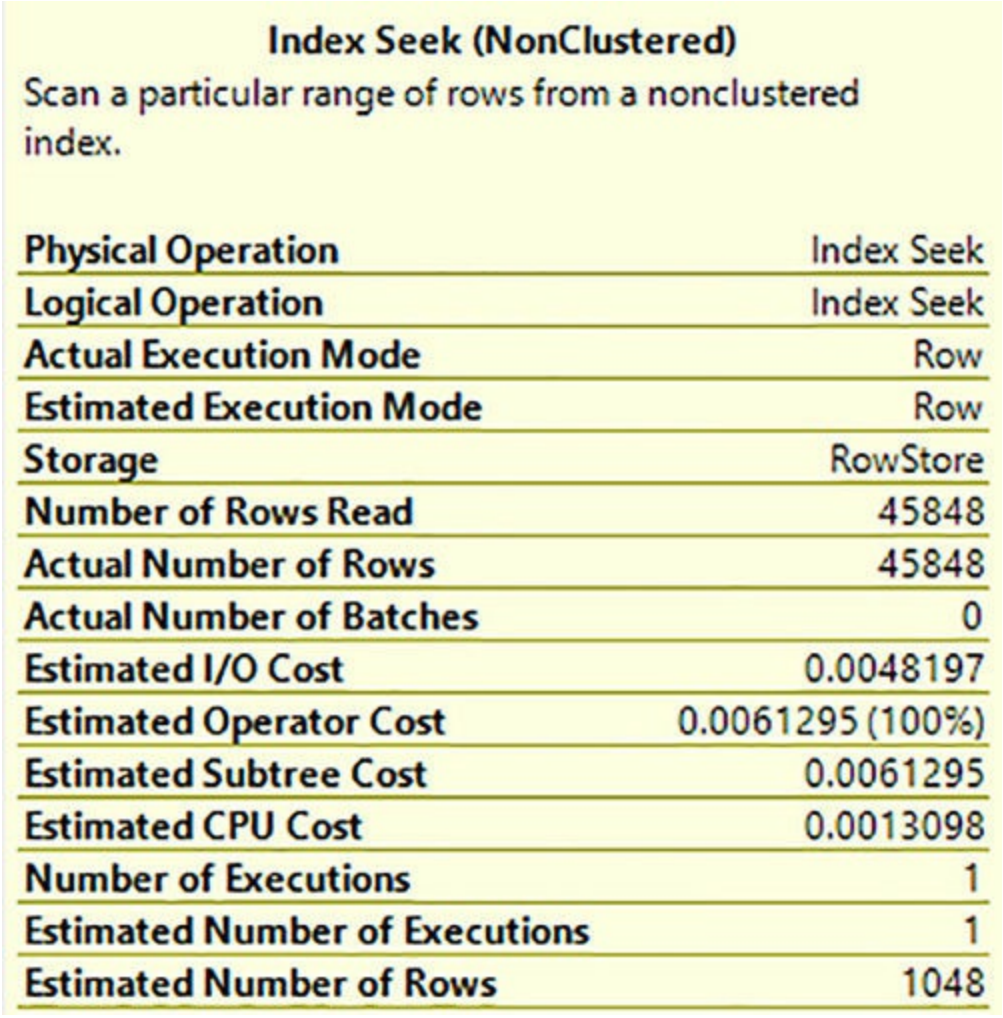
```
UPDATE Examples.OrderLines
    SET StockItemID = 1
    WHERE OrderLineID < 45000;
DBCC SHOW_STATISTICS ('Examples.OrderLines',
    ix_OrderLines_StockItemID );
GO
/* Partial Output
RANGE_HI_KEY   RANGE_ROWS   EQ_ROWS   DISTINCT_RANGE_ROWS   AVG_RAN
-------------- ------------- -------- --------------- ---------
-------
1                  0                 1048                  0
2                  0                 1078                  0
4               1022                 1066                  1
*/
```

Next, click the Include Actual Execution Plan button in the toolbar, and then execute the following query:

```
SELECT StockItemID

FROM Examples.OrderLines

WHERE StockItemID = 1;
```

When you hover the cursor over the Index Seek (NonClustered) in the query plan, notice the difference between Actual Number Of Rows and Estimated Number Of Rows in the tooltip, shown in Figure 4-1. Because the statistics are out-of-date, the estimated row count

is 1048, which is the value currently in the histogram for rows having StockItemID = 1. Because this value is relatively low, the query optimizer generated a plan using an index seek, which could be less optimal than performing a scan when data volumes are high.



Index Seek (NonClustered)

Scan a particular range of rows from a nonclustered index.

Physical Operation	Index Seek
Logical Operation	Index Seek
Actual Execution Mode	Row
Estimated Execution Mode	Row
Storage	RowStore
Number of Rows Read	45848
Actual Number of Rows	45848
Actual Number of Batches	0
Estimated I/O Cost	0.0048197
Estimated Operator Cost	0.0061295 (100%)
Estimated Subtree Cost	0.0061295
Estimated CPU Cost	0.0013098
Number of Executions	1
Estimated Number of Executions	1
Estimated Number of Rows	1048

FIGURE 4-1 Variance between estimated and actual rows in query plan

Review automatic statistics updates

Statistics are generated when you add an index to a table that contains data or when you run the UPDATE STATISTICS command. In most cases, as illustrated by the previous example, you should allow SQL Server to create and update statistics automatically by setting one of the following database options, each of which is enabled by default:

- **AUTO_UPDATE_STATISTICS** SQL Server updates statistics automatically as needed. It determines an update is necessary by using a counter on modifications to column values. This counter is incremented when a row is inserted or deleted or when an indexed column is updated. The counter is reset to 0 when the statistics are generated. When it does this, it acquires compile locks and query plans might require recompilation. You can disable this option by using the sp_autostats system stored procedure.
- **AUTO_UPDATE_STATISTICS_ASYNC** When it is enabled, SQL Server updates

statistics asynchronously. That is, SQL Server uses a background thread so as not to block query execution. In this case, the query optimizer might choose a less than optimal query execution plan until the statistics are updated. Use the ALTER DATABASE T-SQL command to disable this option.

- **AUTO_CREATE_STATISTICS** During query execution, SQL Server creates statistics on individual columns in query predicates to help the query optimizer improve query plans. Use the ALTER DATABASE T-SQL command to disable this option.

Even when statistics are set to update automatically, SQL Server does not update statistics unless one of the following thresholds is met:

- One or more rows is added to an empty table.
- More than 500 rows are added to a table having fewer than 500 rows.
- More than 500 rows are added to a table having more than 500 rows and the number of rows added is more than a dynamic percentage of total rows. With a small table under 25,000 rows, this percentage is around 20 percent. As the number of rows in the table increases, the percentage rate that triggers a statistics update is lower. For example, SQL Server updates statistics for a table with 1 billion rows when more than 1 million changes occur, or 0.1 percent. Prior to SQL Server 2016, this threshold was fixed at 20 percent of the original total number of rows in the table which means that 200 million rows were required to trigger an update of statistics.

You can check to see if SQL Server automatically created statistics in a database by checking the value of the auto_created column in the sys.stats catalog view, as shown in Listing 4-4.

LISTING 4-4 Check auto-created statistics in a database

Click here to view code image

```
Use WideWorldImporters;
GO
SELECT
    OBJECT_NAME(object_id) AS ObjectName,
    name,
    auto_created
FROM sys.stats
WHERE auto_created = 1 AND
    object_id IN
        (SELECT object_id FROM sys.objects WHERE type = 'U');

/* Partial Output
ObjectName                    name                 auto_create
----------------------------------------- ------------------
```

```
-
Colors_Archive                          _WA_Sys_00000001_04E4BC85
OrderLines                              _WA_Sys_00000006_05A3D694
OrderLines                              _WA_Sys_0000000C_05A3D694
plan_persist_runtime_stats              _WA_Sys_00000006_0CBAE877
StockGroups_Archive                    _WA_Sys_00000001_10566F31
StateProvinces                          _WA_Sys_00000002_114A936A
StateProvinces                          _WA_Sys_00000009_114A936A
CustomerTransactions                    _WA_Sys_0000000B_15DA3E5D
*/
```

To check the last time the statistics were updated for each statistics object in a table, you can use the STATS_DATE system function as shown in Listing 4-5. As an alternative, you can use the sys.dm_db_stats_properties DMV to get row counts and modifications occurring since the last statistics update in addition to the last update date.

LISTING 4-5 Check last update of statistics for an object

Click here to view code image

```
SELECT
    name AS ObjectName,
    STATS_DATE(object_id, stats_id) AS UpdateDate
FROM sys.stats
WHERE object_id = OBJECT_ID('Sales.Customers');

/* Output
ObjectName                                                      Up
------------------------------ ------------------------------
--------
PK_Sales_Customers                                  2016-06-02
10:07:35.170
UQ_Sales_Customers_CustomerName                     2016-06-02
10:07:35.240
FK_Sales_Customers_CustomerCategoryID               2016-06-02
10:08:13.080
FK_Sales_Customers_BuyingGroupID                    2016-06-02
10:07:38.010
FK_Sales_Customers_PrimaryContactPersonID           2016-06-02
10:07:43.027
FK_Sales_Customers_AlternateContactPersonID         2016-06-02
10:07:48.040
FK_Sales_Customers_DeliveryMethodID                 2016-06-02
10:07:53.043
FK_Sales_Customers_DeliveryCityID                   2016-06-02
10:07:58.060
```

```
FK_Sales_Customers_PostalCityID                    2016-06-02
10:08:03.060
IX_Sales_Customers_Perf_20160301_06                2016-06-02
10:08:08.067
_WA_Sys_00000003_2FCF1A8A                          2016-06-02
10:10:11.130
_WA_Sys_0000000B_2FCF1A8A                          2016-06-02
10:10:11.623
_WA_Sys_0000001E_2FCF1A8A                          2016-06-02
10:54:31.173
*/
```

Design statistics maintenance tasks

SQL Server creates and updates statistics automatically for all indexes and for columns used in a WHERE or JOIN ON clause. At one extreme, the automatic statistics update process might run when the database is busy and adversely affects performance or, at the other extreme, it might not run frequently enough for a table that is subject to high-volume data changes. For these situations, you can disable the automatic statistics update options for the database and then implement a maintenance plan to update statistics on demand or on a schedule.

> **Note Enabling SQL Server Agent extended stored procedures**
>
> Before you can create a maintenance plan, you must enable SQL Server Agent extended stored procedures. You can do this by starting the SQL Server Agent service in SQL Server Management Studio or by executing the following code:

Click here to view code image

```
EXEC sp_configure 'show advanced options', 1;
GO
RECONFIGURE;
GO
EXEC sp_configure 'Agent XPs', 1;
GO
RECONFIGURE;
GO
```

To create a maintenance plan, open SQL Server Management Studio. and then, in Object Explorer, expand the Management node, right click the Maintenance Plans folder, and then select either New Maintenance Plan or Maintenance Plan Wizard. If you select New Maintenance Plan, type a name for the maintenance plan. Then drag the Update Statistics Task from the Toolbox to the plan designer surface, as shown in Figure 4-2.

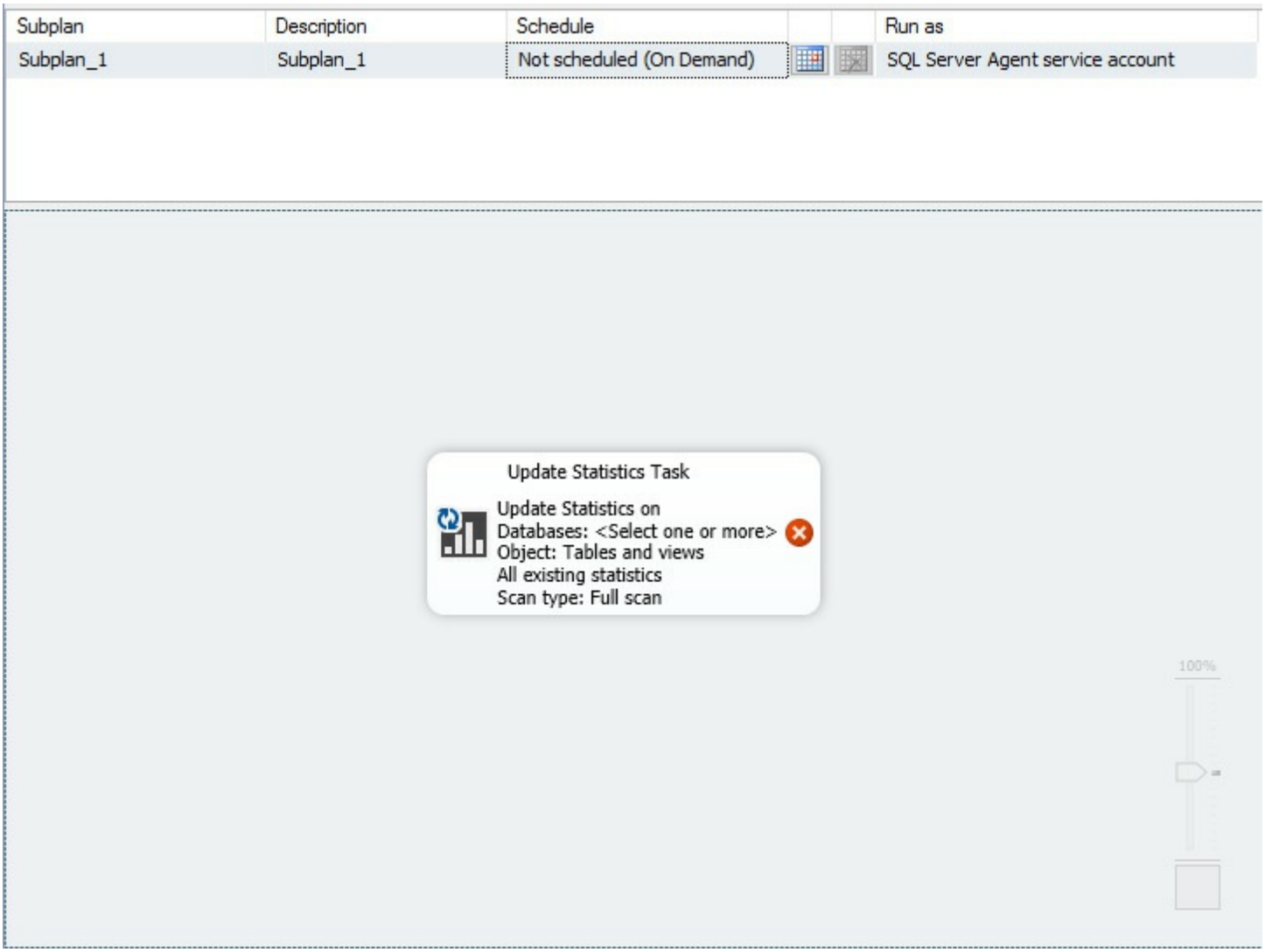


FIGURE 4-2 The Update Statistics Task in the maintenance plan designer

Double click the task to open the Update Statistics Task dialog box, shown in Figure 4-3.

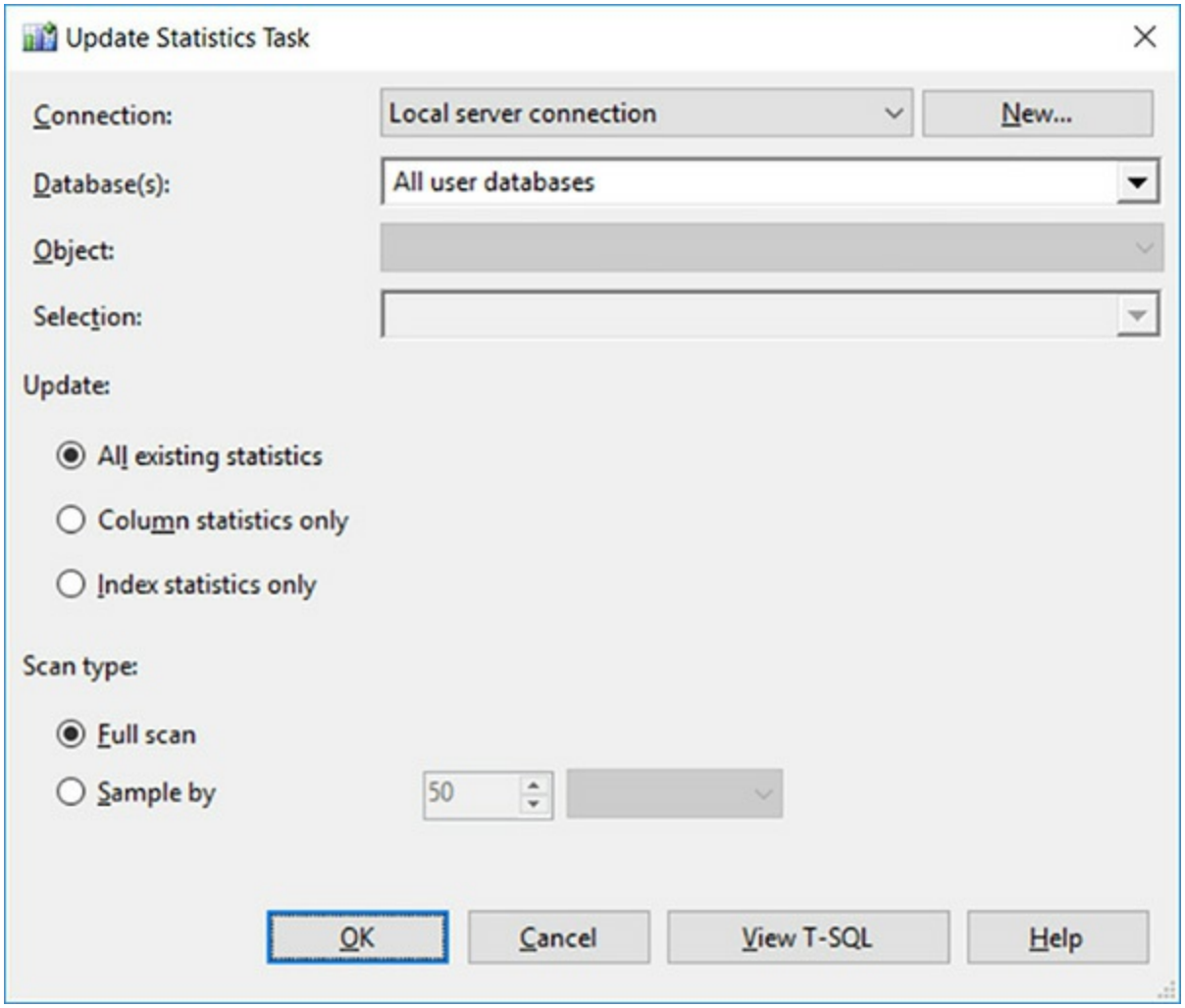


FIGURE 4-3 The Update Statistics Task dialog box

In the Databases drop-down list, you can select one of the following options:

- **All Databases** All SQL Server databases, except tempdb.
- **System Databases** All SQL Server system databases, except tempdb.
- **All User Databases** (Excluding Master, Model, Msdb, Tempdb) All user databases and none of the SQL Server system databases.
- **These Databases** A list of user and SQL Server system databases (except tempdb) from which you must select at least one database. If you select this option, you must also specify whether to update statistics for tables, views, or both tables and views. If you select Tables or Views, you must also select one or more tables or views from the respective list.

Then you specify one of the following options for the update:

- **All Existing Statistics** Statistics for both columns and indexes.
- **Column Statistics Only** Statistics for columns only.
- **Index Statistics Only** Statistics for indexes only.

Last, you select one of the following options for scan type:

- **Full Scan** Update statistics by reading all rows in a table or view.
- **Sample By** Update statistics based on a specified percentage or specified number of rows. This is a better option when updating statistics for a large table or view.

You can click the View T-SQL button at the bottom of the Update Statistics Task dialog box to generate the T-SQL script for the new maintenance task. Then you can use this script as a template to create additional maintenance plans for updating statistics. Listing 4-6 shows a portion of the script generated for the WideWorldImporters database:

LISTING 4-6 Script to update statistics for a specific table

Click here to view code image

```
USE WideWorldImporters;
GO
UPDATE STATISTICS [Application].[Cities]
WITH FULLSCAN
GO
```

Note UPDATE STATISTICS options

You can add any of the following options when using the UPDATE STATISTICS statement.

- No FULLSCAN or SAMPLE option If you omit the FULLSCAN or SAMPLE option, SQL Server calculates statistics by computing an appropriate sample size and performing a sample scan.
- FULLSCAN SQL Server performs a full scan of the table data or the index to generate more accurate statistics, although this option takes more time and more IO.
- SAMPLE With this option, you specify the number or percentage of rows that SQL Server samples when generating the statistics.
- RESAMPLE SQL Server generates the statistics using the same sampling ratio that was defined during the previous statistics generation.

The maintenance task should be scheduled at a time that interferes least with regular database operations. On the Maintenance Plan designer, click the Subplan Schedule button to open the New Job Schedule dialog box. Here you can specify whether the maintenance plan is a recurring schedule, whether it runs only when SQL Server starts or when the CPU is idle, or one time only. If you choose the Recurring option, you specify a frequency, and a start and optional end date. If you instead choose the One Time option, you specify the date and time to execute the maintenance plan. When you save the schedule, a SQL Server Agent

job is created and the job executes as a SQL Server Integration Services (SSIS) package.

You can also execute a maintenance plan without waiting for the scheduled SQL Server Agent job. To do this, expand the Management node, expand the Maintenance Plan node, right click the maintenance plan, and then select Execute. When you use this method, the SSIS package is launched and you can observe the execution status in the Execute Maintenance Plan dialog box.

Use dynamic management objects to review current index usage and identify missing indexes

SQL Server uses indexes to speed up data access. In Chapter 1, we covered many of the considerations affecting the design of tables. Over time, you might find that some indexes are not as helpful as expected due to changes in the distribution of the data or in the query patterns. Furthermore, the existence of an index that SQL Server never uses adds overhead to write operations. Therefore, it's important to periodically review indexes not only to determine whether existing indexes are still useful, but also whether any are ignored or missing.

In this section, we review several dynamic management objects (DMOs) that are useful for this review process. We focus on how to perform specific review tasks that are important for index management without delving into all the possible information that you can derive from using these DMOs.

> **Need More Review? Resources for managing indexes with DMOs**
>
> You can find more detail about each DMO by accessing the respective topic in Books Online. For deeper coverage of index management by using DMOs, download the free ebook "Performance Tuning with Dynamic Management Views" by Tim Ford and Louis Davidson available at *https://www.simple-talk.com/books/sql-books/performance-tuning-with-sql-server-dynamic-management-views---ebook-download*.

Review current index usage

The following DMOs provide information about whether and how indexes are used:

- **sys.dm_db_index_usage_stats** Use this DMV to review the use of indexes to resolve queries.
- **sys.dm_db_index_physical_stats** Use this dynamic management function (DMF) to check the overall status of indexes in a database.

sys.dm_db_index_usage_stats

To get a quick overview of which indexes are being used, you can use the sys.dm_db_index_usage_stats DMV as shown in Listing 4-7. To appear in this DMV's

output, an index must be read or written to at least once by a user or system operation. In this example, the count of user seeks, scans, and lookups are aggregated as user_reads and sorted in descending order to make it clear which indexes are used more frequently than others. Counts in this DMV are reset when the server restarts or when an index is dropped and recreated.

LISTING 4-7 Review current index usage

Click here to view code image

```
SELECT
    OBJECT_NAME(ixu.object_id, DB_ID('WideWorldImporters')) AS
[object_name] ,
    ix.[name] AS index_name ,
    ixu.user_seeks + ixu.user_scans + ixu.user_lookups AS
user_reads,
    ixu.user_updates AS user_writes
FROM sys.dm_db_index_usage_stats ixu
INNER JOIN WideWorldImporters.sys.indexes ix ON
    ixu.[object_id] = ix.[object_id] AND
    ixu.index_id = ix.index_id
WHERE ixu.database_id = DB_ID('WideWorldImporters')
ORDER BY user_reads DESC;
/*Partial Output
object_name                index_name                     user_rea
user_writes
---------------------------------------- --------------- ---
-----------
StockGroups                    PK_Warehouse_StockGroups    19
Suppliers                      PK_Purchasing_Suppliers     13
ColdRoomTemperatures_Archive   ix_ColdRoomTemperatures_Archive
StockItems                     FK_Warehouse_StockItems_SupplierI
*/
```

Just as important as knowing which indexes are being accessed is knowing which indexes are never used. To find these indexes, you must start by retrieving all tables and indexes in a database and then filter out those appearing in sys.dm_db_index_usage_stats as shown in Listing 4-8.

LISTING 4-8 Find unused indexes

Click here to view code image

```
USE WideWorldImporters;
GO
```

```
SELECT
    OBJECT_NAME(ix.object_id) AS ObjectName ,
    ix.name
FROM sys.indexes AS ix
INNER JOIN sys.objects AS o ON
    ix.object_id = o.object_id
WHERE ix.index_id NOT IN (
    SELECT ixu.index_id
    FROM sys.dm_db_index_usage_stats AS ixu
    WHERE
        ixu.object_id = ix.object_id AND
        ixu.index_id = ix.index_id AND
        database_id = DB_ID()
    ) AND
    o.[type] = 'U'
ORDER BY OBJECT_NAME(ix.object_id) ASC ;

/* Partial Output
ObjectName                          name
-------------------------------------------------------------
---------------
BuyingGroups            UQ_Sales_BuyingGroups_BuyingGroupName
BuyingGroups_Archive    ix_BuyingGroups_Archive
Cities_Archive          ix_Cities_Archive
ColdRoomTemperatures    NULL
ColdRoomTemperatures    PK_Warehouse_ColdRoomTemperatures
ColdRoomTemperatures    IX_Warehouse_ColdRoomTemperatures_ColdR
*/
```

Whereas the previous example helps you find indexes for which there is no read or write activity, Listing 4-9 helps you find indexes that SQL Server maintains but never uses to retrieve data for a query. These indexes are consuming resources without helping query performance. You should consider dropping these indexes if further investigation reveals that there is no need to continue to maintain them.

LISTING 4-9 Find indexes that are updated but never used

Click here to view code image

```
USE WideWorldImporters;
GO
SELECT
    o.name AS ObjectName ,
    ix.name AS IndexName ,
    ixu.user_seeks + ixu.user_scans + ixu.user_lookups AS
user_reads ,
```

```
    ixu.user_updates AS user_writes ,
    SUM(p.rows) AS total_rows
FROM sys.dm_db_index_usage_stats ixu
INNER JOIN sys.indexes ix ON
    ixu.object_id = ix.object_id AND
    ixu.index_id = ix.index_id
INNER JOIN sys.partitions p ON
    ixu.object_id = p.object_id AND
    ixu.index_id = p.index_id
INNER JOIN sys.objects o ON
    ixu.object_id = o.object_id
WHERE
    ixu.database_id = DB_ID() AND
    OBJECTPROPERTY(ixu.object_id, 'IsUserTable') = 1 AND
    ixu.index_id > 0
GROUP BY
    o.name ,
    ix.name ,
    ixu.user_seeks + ixu.user_scans + ixu.user_lookups ,
    ixu.user_updates
HAVING ixu.user_seeks + ixu.user_scans + ixu.user_lookups = 0
ORDER BY
    ixu.user_updates DESC,
    o.name ,
    ix.name ;

/* Output
ObjectName                    IndexName    user_reads  user_writ
----------------------------- ---------------- ---------------- --
---------------
ColdRoomTemperatures_Archive
ix_ColdRoomTemperatures_Archive  0  2016  3654736
StockItems                    FK_Warehouse_StockItems_SupplierID
0   0      227
*/
```

---

sys.dm_db_index_physical_stats

In addition to reviewing usage of indexes, you should also review index health by using the sys.dm_db_index_physical_stats DMF. As inserts, updates, and deletes occur, an index becomes increasingly fragmented and IO increases as data is no longer efficiently stored on disk. Listing 4-10 shows how to review fragmentation. In general, you should focus on indexes for which fragmentation is greater than 15percent and the page count is greater than 500. When fragmentation is between 15 percent and 30 percent, you should reorganize the index, and when its greater, you should rebuild it.

LISTING 4-10 Review index fragmentation

Click here to view code image

```
DECLARE  @db_id SMALLINT, @object_id INT;
SET @db_id = DB_ID(N'WideWorldImporters');
SET @object_id = OBJECT_ID(N'WideWorldImporters.Sales.Orders');
SELECT
    ixs.index_id AS idx_id,
    ix.name AS ObjectName,
    index_type_desc,
    page_count,
    avg_page_space_used_in_percent AS AvgPageSpacePct,
    fragment_count AS frag_ct,
    avg_fragmentation_in_percent AS AvgFragPct
FROM sys.dm_db_index_physical_stats
    (@db_id, @object_id, NULL, NULL , 'Detailed') ixs
INNER JOIN sys.indexes ix ON
    ixs.index_id = ix.index_id AND
    ixs.object_id = ix.object_id
ORDER BY avg_fragmentation_in_percent DESC;

/* Output
idx_id ObjectName   index_type_desc    pg_ct
AvgPageSpacePct   frag_ct  AvgFragPct
-------- ---------------------------------- ---------- --------
------------------
4 FK_Sales_Orders_PickedByPersonID  NONCLUSTERED
INDEX  237  53.6864838151717 237

2 FK_Sales_Orders_CustomerID        NONCLUSTERED INDEX   189
67.3274277242402 189

5 FK_Sales_Orders_ContactPersonID   NONCLUSTERED
INDEX  189  67.3274277242402 189

3 FK_Sales_Orders_SalespersonPersonID NONCLUSTERED INDEX 136
93.5749444032617 136

1 PK_Sales_Orders                    CLUSTERED INDEX    688
99.0945268099827   88

1 PK_Sales_Orders                    CLUSTERED
INDEX    2   55.2260934025204  2 0
1 PK_Sales_Orders                    CLUSTERED INDEX    1
0.296515937731653  1  0
2 FK_Sales_Orders_CustomerID         NONCLUSTERED INDEX  1
```

```
39.6713615023474 1   0
3 FK_Sales_Orders_SalespersonPersonID NONCLUSTERED INDEX  1
28.5396590066716 1  0
4 FK_Sales_Orders_PickedByPersonID    NONCLUSTERED INDEX  1
58.5001235483074 1  0
5 FK_Sales_Orders_ContactPersonID     NONCLUSTERED INDEX  1
39.6713615023474 1  0
*/
```

Identify missing indexes

When the query optimizer compiles a T-SQL statement, it also tracks up to 500 indexes that could have been used if they had existed. The following DMVs help you review these missing indexes:

- **sys.dm_db_missing_index_details** Use this DMV to identify the columns used for equality and inequality predicates.
- **sys.dm_db_missing_index_groups** Use this DMV as an intermediary between sys.dm_db_index_details and sys.dm_db_missing_group_stats.
- **sys.dm_db_missing_index_group_stats** Use this DMV to retrieve metrics on a group of missing indexes.

> **Note SQL Server restart clears information from DMOs**
>
> The information in these DMOs is cleared when SQL Server restarts and reflects only information accumulated since the last restart.

You use the three DMVs as a group, as shown in Listing 4-11. The output, which will vary on your computer, will list each database, schema, and table that is missing an index in descending order of the overall improvement expected by adding an index. The improvement is derived by multiplying the sum of the seeks and scans that the index helps by the average cost of the user queries that could be reduced by the index and by the average percent decrease in cost resulting from implementing the index. The equality_columns column lists one or more columns in the table that are good candidates for the new index. The inequality_columns column lists columns that are useful for queries that include the <> operator that you might consider adding to the index. Last, the included_columns column lists the columns that are suggested for addition to the index in the INCLUDE clause.

LISTING 4-11 Review missing indexes

Click here to view code image

```
SELECT
    (user_seeks + user_scans) * avg_total_user_cost *
(avg_user_impact * 0.01) AS
IndexImprovement,
    id.statement,
    id.equality_columns,
    id.inequality_columns,
    id.included_columns
FROM sys.dm_db_missing_index_group_stats AS igs
INNER JOIN sys.dm_db_missing_index_groups AS ig
    ON igs.group_handle = ig.index_group_handle
INNER JOIN sys.dm_db_missing_index_details AS id
    ON ig.index_handle = id.index_handle
ORDER BY IndexImprovement DESC;

/* Output
IndexImprovmeent  statement
equality_columns   inequality_columns  included_columns
-------------------------- ------------------------------ -----
--------------------
79.89008274829 [AdventureWorksDW].[dbo].[FactInternetSales]
[ProductKey] NULL

*/
```

Consolidate overlapping indexes

Consider a situation in which two indexes include similar indexes such as those shown in Listing 4-12. In this example, the two indexes are the same except one index contains an additional column. When SQL Server processes a query that needs the columns in the smaller index, it uses the smaller index. Conversely, when processing a query that uses all columns in the larger index, then SQL Server uses that index.

LISTING 4-12 Create overlapping indexes

Click here to view code image

```
USE [WideWorldImporters];
GO
CREATE NONCLUSTERED INDEX [IX_Sales_Invoices_ExamBook762Ch4_A]
    ON [Sales].[Invoices]
(
    [CustomerID],
    [InvoiceDate]
)
INCLUDE ([TotalDryItems]);
```

```
GO
CREATE NONCLUSTERED INDEX [IX_Sales_Invoices_ExamBook762Ch4_B]
    ON [Sales].[Invoices]
(
    [CustomerID],
    [InvoiceDate],
    [CustomerPurchaseOrderNumber]
)
INCLUDE ([TotalDryItems]);
GO
```

With regard to query performance, all is well. However, SQL Server has an additional index to manage and requires more disk space for the database when similar indexes exist. This situation can occur when you create new indexes based on the DMVs for missing indexes or follow the recommendations from the Database Engine Tuning Advisor because they do not always account for existing indexes. Therefore, when using the DMVs or Database Engine Tuning Advisor to identify missing indexes, you might consider simply adding a new column to an existing index by adding it as another key or as an included column.

Meanwhile, when you encounter overlapping indexes, you should drop one of them so that database maintenance tasks run faster and less storage is required. Use the script in Listing 4-13 to find overlapping indexes.

Exam Tip

The exam has several different styles of questions to test your understanding of missing indexes and overlapping indexes. In particular, you should know how to identify overlapping indexes and what actions are necessary to consolidate and optimize them. It's also important to understand which operators can be removed with a clustered or covering index.

LISTING 4-13 Find overlapping indexes

Click here to view code image

```
USE [WideWorldImporters];
WITH IndexColumns AS (
    SELECT
        '[' + s.Name + '].[' + T.Name + ']' AS TableName,
        ix.name AS IndexName,
        c.name AS ColumnName,
        ix.index_id,
        ixc.index_column_id,
```

```
        COUNT(*) OVER(PARTITION BY t.OBJECT_ID, ix.index_id) AS
ColumnCount
    FROM sys.schemas AS s
    INNER JOIN sys.tables AS t ON
        t.schema_id = s.schema_id
    INNER JOIN sys.indexes AS ix ON
        ix.OBJECT_ID = t.OBJECT_ID
    INNER JOIN sys.index_columns AS ixc ON
        ixc.OBJECT_ID = ix.OBJECT_ID AND
        ixc.index_id = ix.index_id
    INNER JOIN sys.columns AS c ON
        c.OBJECT_ID = ixc.OBJECT_ID AND
        c.column_id = ixc.column_id
WHERE
        ixc.is_included_column = 0 AND
        LEFT(ix.name, 2) NOT IN ('PK', 'UQ', 'FK')
)
SELECT DISTINCT
    ix1.TableName,
    ix1.IndexName AS Index1,
    ix2.IndexName AS Index2
FROM IndexColumns AS ix1
INNER JOIN IndexColumns AS ix2 ON
    ix1.TableName = ix2.TableName AND
    ix1.IndexName <> ix2.IndexName AND
    ix1.index_column_id = ix2.index_column_id AND
    ix1.ColumnName = ix2.ColumnName AND
    ix1.index_column_id < 3 AND
    ix1.index_id < ix2.index_id AND
    ix1.ColumnCount <= ix2.ColumnCount
ORDER BY ix1.TableName, ix2.IndexName;

/* Output
TableName        Index1                                          In
-------------------------------------------------------------------
----
      [Sales].[Invoices] IX_Sales_Invoices_ExamBook762Ch4_A
                         IX_Sales_Invoices_ExamBook762Ch4_B
[Sales].[OrderLines]IX_Sales_OrderLines_AllocatedStockItems
                       IX_Sales_OrderLines_Perf_20160301_02
*/
```

Skill 4.2: Analyze and troubleshoot query plans

One of the most important skills that you can have as a database administrator is the ability to analyze and troubleshoot query plans. In this section, we explain how to capture query plans, how certain query plan operators can indicate a potential performance problem, and

how to interpret estimated versus actual query plans. We also introduce Query Store as a SQL Server 2016 feature that you can use to review SQL Server's selection of query plans for a query over time. For Azure SQL Database implementations, we show you how Azure SQL Database Performance Insight provides visibility into query performance in the cloud.

This section covers how to:

- Capture query plans using extended events and traces
- Identify poorly performing query plan operators
- Create efficient query plans using Query Store
- Compare estimated and actual query plans and related metadata
- Configure Azure SQL Database Performance Insight

Capture query plans using extended events and traces

Before a query executes, several processes occur to manage how SQL Server performs the instructions in the T-SQL statement. The first process is *query parsing*, a step in which the database engine checks to make sure the submitted query uses valid T-SQL syntax. If query parsing is successful and if the T-SQL statement is a DML statement, the next process to run is the *algebrizer*, which verifies the existence of the referenced objects, such as tables and columns. If this process succeeds, the next process invoked is the *query optimizer*. The query optimizer checks to see if a query plan already exists for the query. If not, it generates one or more query plans based on the statistics available for the data and then selects the query plan that is good enough to perform the task at hand while minimizing the use CPU and IO when possible.

Although you can use the graphical query plan to analyze a single query, such as you might during index design as described in Chapter 1, you need an alternate approach when you need to troubleshoot many queries running on a server. In that case, you can automate the process of capturing execution plans by using extended events or SQL Trace.

Important Permissions required to view a query plan

You must have the appropriate permissions within a database to view a query plan. If your login is assigned to the sysadmin, dbcreator, or db_owner role, you have the necessary permissions. Otherwise, you need to be granted the SHOWPLAN permission by running the following statement:

```
GRANT SHOWPLAN TO [username];
```

Extended Events

Using Extended Events is a lightweight approach to capturing query plans. There are two Extended Events that you can use to review query plans:

- **query_pre_execution_showplan** This Extended Event captures the estimated query plan for a query. An estimated query plan is prepared without executing the query.
- **query_post_execution_showplan** This Extended Event captures the actual query plan for a query. An actual query plan is the estimated query plan that includes runtime information. For this reason, it is not available until after the query executes.

Listing 4-14 shows how to create and start an Extended Event session for actual query plans. In this example, the session definition filters the query activity on the server for a specific database and query type, ADHOC. In your own environment, you can remove filters entirely or apply more filters as needed. We describe how to work with Extended Events in greater detail in Skill 4.4.

Note Create target folder for query plan before running sample script

The target folder for the query plan, C:\ExamBook762Ch4\, must exist before running the script shown in Listing 4-14.

Also, be aware that running an Extended Event session to capture actual query plans is an expensive operation and should be used sparingly on a production server and only with highly selective filtering in place.

LISTING 4-14 Create and start an Extended Event session to capture an actual query plan

Click here to view code image

```
IF EXISTS(SELECT *
    FROM sys.server_event_sessions
    WHERE name='ActualQueryPlans')
    DROP EVENT SESSION ActualQueryPlans
    ON SERVER;
GO
CREATE EVENT SESSION ActualQueryPlans
ON SERVER
ADD EVENT sqlserver.query_post_execution_showplan(
    ACTION (sqlserver.database_name,
                    sqlserver.client_hostname,
                    sqlserver.client_app_name,
                    sqlserver.plan_handle,
                    sqlserver.sql_text,
                    sqlserver.tsql_stack,
                    package0.callstack,
                    sqlserver.query_hash,
                    sqlserver.session_id,
                    sqlserver.request_id)
    WHERE
```

```
        sqlserver.database_name='WideWorldImporters' AND
        object_type = 'ADHOC'
)
ADD TARGET package0.event_file(SET
filename=N'C:\ExamBook762Ch4\ActualQueryPlans.xel',
    max_file_size=(5),max_rollover_files=(4)),
ADD TARGET package0.ring_buffer
    WITH (MAX_DISPATCH_LATENCY=5SECONDS, TRACK_CAUSALITY=ON);
GO
ALTER EVENT SESSION ActualQueryPlans
    ON SERVER
    STATE=START;
GO
```

You can review the query plans captured by this Extended Event using the graphical interface in SQL Server Management Studio. In Object Explorer, expand the Management node, expand the Sessions node, right click ActualQueryPlans, and select Watch Live Data. Now that you are watching the sessions, execute a query, like this:

```
USE WideWorldImporters;
GO
SELECT *
FROM Warehouse.StockGroups;
```

In the session window, click the row in which the query event appears, and then click the Query Plan tab in the lower portion of screen, as shown in Figure 4-4.

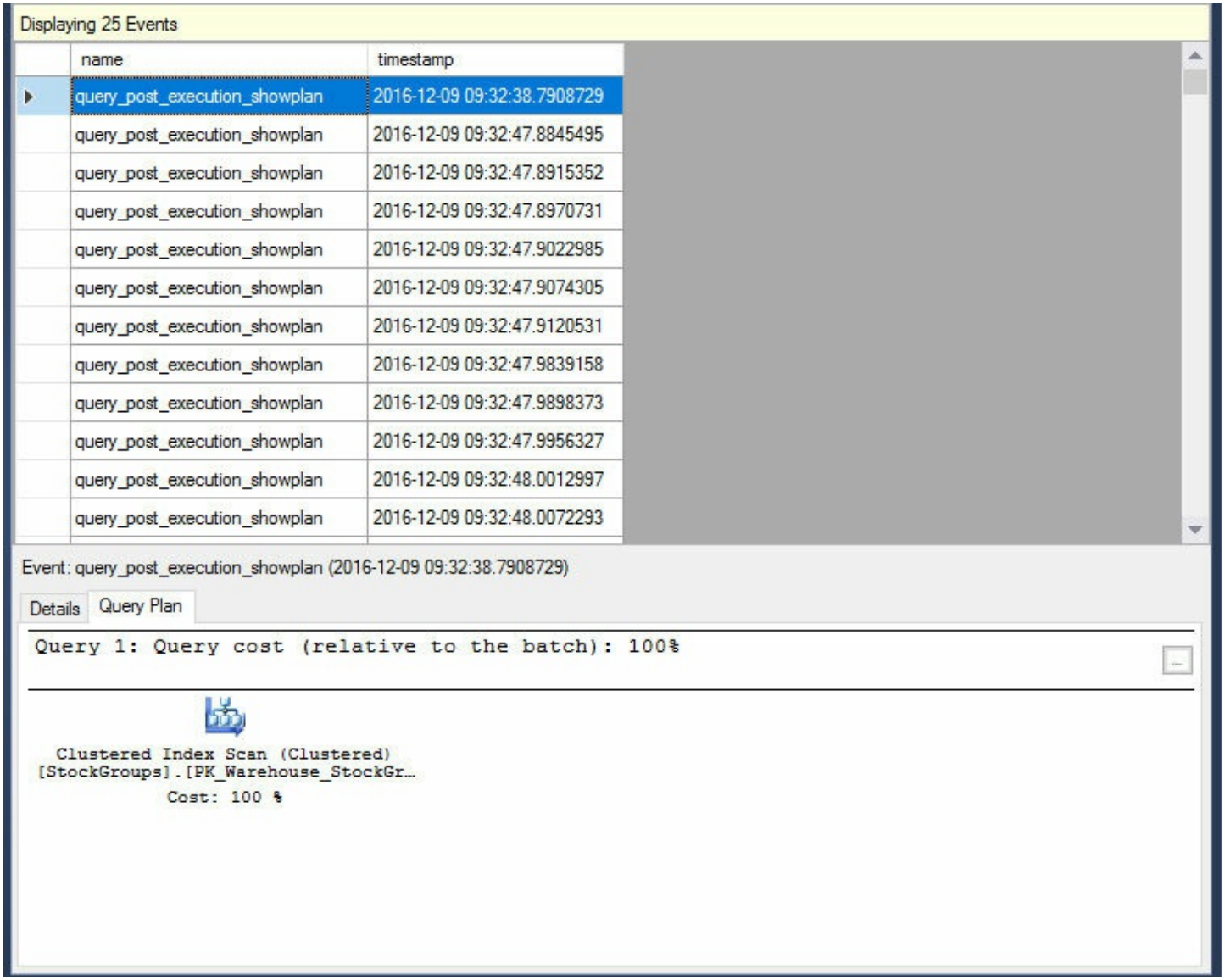


FIGURE 4-4 A query plan accessed from an Extended Event session

When you no longer need to capture query plans, be sure to disable or drop the Extended Event session, using the applicable statement in Listing 4-15.

LISTING 4-15 Disable or drop extended event sessions

Click here to view code image

```
--Disable extended event session
ALTER EVENT SESSION ActualQueryPlans
    ON SERVER
    STATE=STOP;
GO
--Drop extended event session
IF EXISTS(SELECT *
    FROM sys.server_event_sessions
    WHERE name='ActualQueryPlans')
```

```
    DROP EVENT SESSION ActualQueryPlans
    ON SERVER;
GO
```

SQL Trace

Although SQL Trace is designated as a deprecated feature and will be removed from a future release of SQL Server, it remains an available option in SQL Server 2016. You can define server-side traces by using system stored procedures and then run these traces on demand or on a scheduled basis. As an alternative, you can use SQL Server Profiler as a client-side option. The overhead of running server-side traces is much less than the overhead of using SQL Server Profiler, but the overhead is still significant. Therefore, take care when using SQL Trace in a production environment regardless of the approach you take and disable tracing as soon as possible.

Server-side tracing

To define a trace, use the following system stored procedures:

- **sp_trace_create** This procedure creates a new trace and defines a file into which SQL Server stores trace data. It returns a trace ID that you reference in the other procedures to manage the trace.
- **sp_trace_setevent** This procedure must be called once for each data column of the events to capture in the trace. That means you must call this procedure many times for any single trace. When you call this procedure, you pass in the following arguments, the trace identifier captured as output when you create the trace, the event identifier, the column identifier, and the status of ON (1) or OFF (0).
- **sp_trace_setfilter** This procedure must be called once for each filter on an event data column.
- **sp_trace_setstatus** This procedure starts, stops, or removes a trace. It must be stopped and removed before you can open the related trace file.

Listing 4-16 illustrates how to use these four system stored procedures to create a trace for a query plan. The trace data is stored in the ExamBook762Ch4 folder that must exist prior to executing the sp_trace_create system stored procedure. Next, the sp_trace_setevent system stored procedure is called multiple times to capture the query plan, the login name associated with the query, the start and end time of the query, and the text of the query, respectively.

LISTING 4-16 Create a trace, add events and filter to a trace, and start a trace

Click here to view code image

```
USE master;
```

```
GO
DECLARE @TraceID int;
EXEC sp_trace_create
    @TraceID output,
    0,
    N'C:\ExamBook762Ch4\ActualQueryPlanTrc';

EXEC sp_trace_setevent @TraceID,
    146,    -- Showplan XML Statistics Profile
    27,     -- BinaryData column
    1;      -- Column is ON for this event

EXEC sp_trace_setevent @TraceID,
    146,
    1,       -- TextData column
    1;

EXEC sp_trace_setevent @TraceID,
    146,
    14,      -- StartTime column
    1;

EXEC sp_trace_setevent @TraceID,
    146,
    15,      -- EndTime column
    1;

-- Set filter for database
EXEC sp_trace_setfilter @TraceID,
    @ColumnID = 35, --Database Name
    @LogicalOperator = 0, -- Logical AND
    @ComparisonOperator = 6, -- Comparison LIKE
    @Value = N'WideWorldImporters' ;

-- Set filter for application name
EXEC sp_trace_setfilter @TraceID,
    @ColumnID = 10, --ApplicationName
    @LogicalOperator = 0, -- Logical AND
    @ComparisonOperator = 6, -- Comparison LIKE
    @Value = N'Microsoft SQL Server Management Studio - Query'
;

-- Start Trace (status 1 = start)
EXEC sp_trace_setstatus @TraceID, 1;
GO
```

Note Resources for sp_trace_setevent and sp_trace_setfilter

Lists of event and column identifiers for defining the event columns are available at "sp_trace_setevent (Transact-SQL)", *https://msdn.microsoft.com/en-us/library/ms186265.aspx*. Lists of the logical and comparison operators for the trace filter are available at "sp_trace_setfilter (Transact-SQL)" at *https://msdn.microsoft.com/en-us/library/ms174404.aspx*.

Now execute a query to generate an event for the trace like this:

```
USE WideWorldImporters;
GO
SELECT *
FROM Warehouse.StockGroups;
```

To view the trace information, find the trace identifier by using the sys.fn_trace_getinfo system function and then use sp_trace_setstatus twice, as shown in Listing 4-17, replacing <traceid> with the trace identifier that you find by executing the first statement. The first time you set the status of the trace to 0 to stop it and the second time you set the status to 2 to close and delete the trace information from SQL Server.

LISTING 4-17 Stop and delete a trace

Click here to view code image

```
----  Find the trace ID
USE master;
GO
SELECT *
FROM sys.fn_trace_getinfo(0)
WHERE value = 'C:\ExamBook762Ch4\ActualQueryPlanTrc.trc';

-- Set  the trace status to stop
EXEC sp_trace_setstatus
    @traceid = <traceid>,
    @status= 0;
GO

-- Close and Delete the trace
EXEC sp_trace_setstatus
    @traceid = <traceid>,
    @status = 2;
GO
```

The trace file remains on the file system and is available for you to view in SQL Server Profiler. Be sure to open SQL Server Profiler by using the Run As Administrator option and then open the trace file. Click the row containing the query's Showplan event to view the graphical query plan, as shown in Figure 4-5.

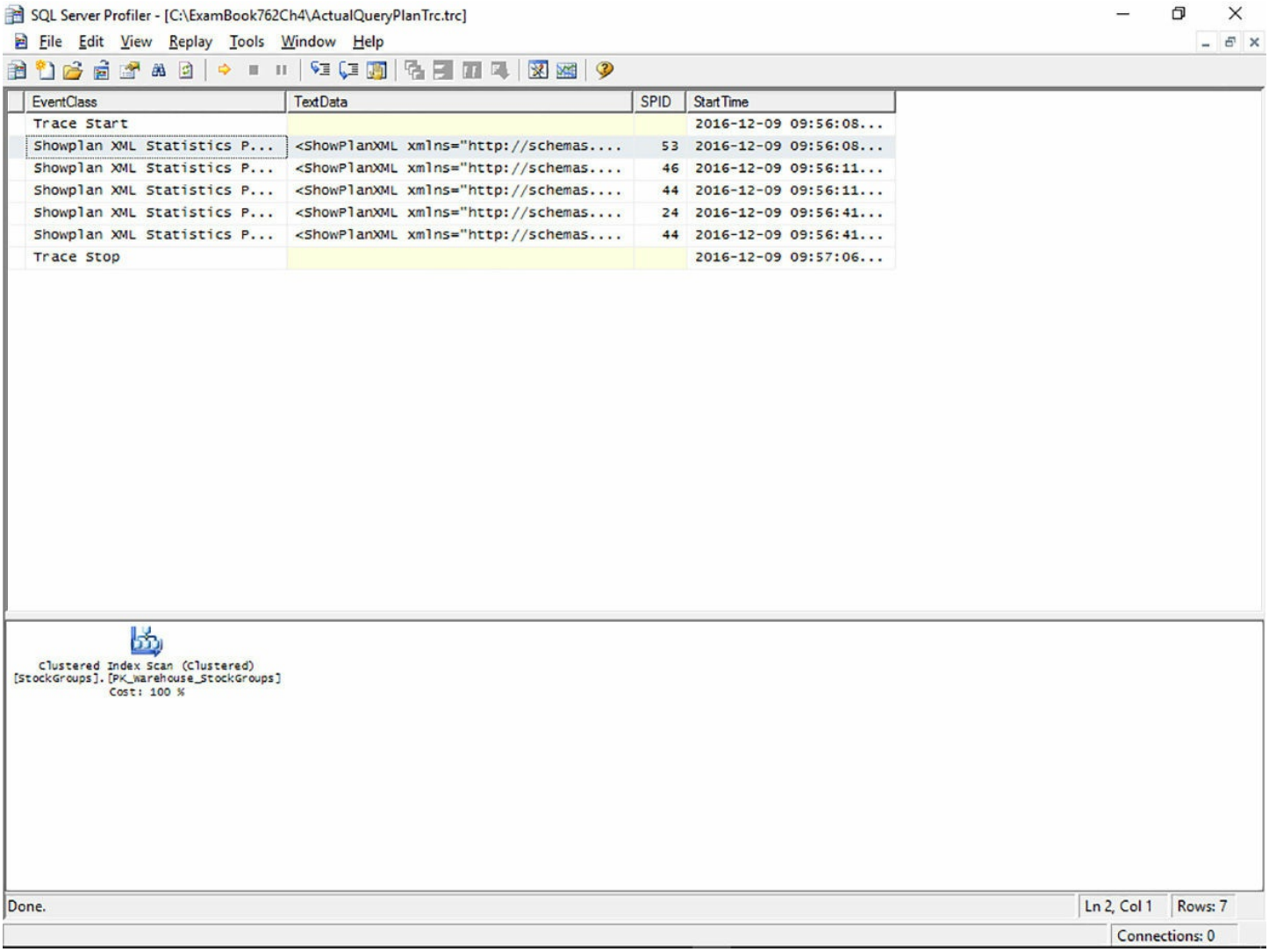


FIGURE 4-5 A query plan accessed from a SQL trace file

Setting up traces manually by using the system stored procedures can be tedious due to the number of numeric parameters required. Rather than refer to Books Online to find the necessary values, you can take advantage of SQL Server catalog views to find the values you need, as shown in Listing 4-18.

LISTING 4-18 Get event and column identifiers for use in a trace definition

Click here to view code image

```
--Get event identifiers
SELECT
    e.trace_event_id AS EventID,
    e.name AS EventName,
```

```
    c.name AS CategoryName
FROM sys.trace_events e
JOIN sys.trace_categories c
    ON e.category_id = c.category_id
ORDER BY e.trace_event_id;

/* Partial output
EventID    EventName                      CategoryName
---------- ------------------------------ ------------------
---
10             RPC:Completed                   Stored
Procedures
11             RPC:Starting                   Stored
Procedures
12             SQL:BatchCompleted     TSQL
13             SQL:BatchStarting           TSQL
*/

--Get column identifiers for events
SELECT
    trace_column_id,
    name AS ColumnName
FROM sys.trace_columns
ORDER BY trace_column_id;

/* Partial output
trace_column_id  ColumnName
---------------------- ----------------------------
1                              TextData
2                              BinaryData
3                              DatabaseID
4                              TransactionID
5                              LineNumber
*/
```

LISTING 4-19 Create an indexed view to improve aggregate query performance

Click here to view code image

```
CREATE VIEW Sales.vSalesByYear
WITH SCHEMABINDING
AS
    SELECT
    YEAR(InvoiceDate) AS InvoiceYear,
    COUNT_BIG(*) AS InvoiceCount
FROM Sales.Invoices
```

```
GROUP BY YEAR(InvoiceDate);
GO
CREATE UNIQUE CLUSTERED INDEX idx_vSalesByYear
    ON Sales.vSalesByYear
    (InvoiceYear);
GO
```

Client-side tracing

You can use SQL Server Profiler instead of manually creating the many stored procedures to define a trace when you need to capture a query plan. On the File menu, click New Trace, and then connect to the server on which you want to run the trace. In the Trace Properties dialog box, click the Events Selection tab and then select the Show All Events checkbox. Expand the Performance node, and select one or more of the following checkboxes:

- **Showplan XML** This event is raised when SQL Server selects an estimated query plan.
- **Showplan XML For Query Compile** This event is raised when SQL Server compiles a query and produces an estimated query plan which it adds to the query plan cache. Generally, this event is raised only once for a query unless the query requires recompilation.
- **Showplan XML Statistics Profile** This event is raised after SQL Server executes a query and has generated an actual query plan.

To minimize the performance impact of running SQL Server Profiler, you should apply as many filters as possible to capture only the query plans of interest. As an example, you might want to focus on queries in a specific database. To do this, select the Show All Columns checkbox, and then click Column Filters. In the Edit Filter dialog box, click DatabaseName, expand Like, type WideWorldImporters as shown in Figure 4-6, and then click OK. Next, click Run to start the trace.

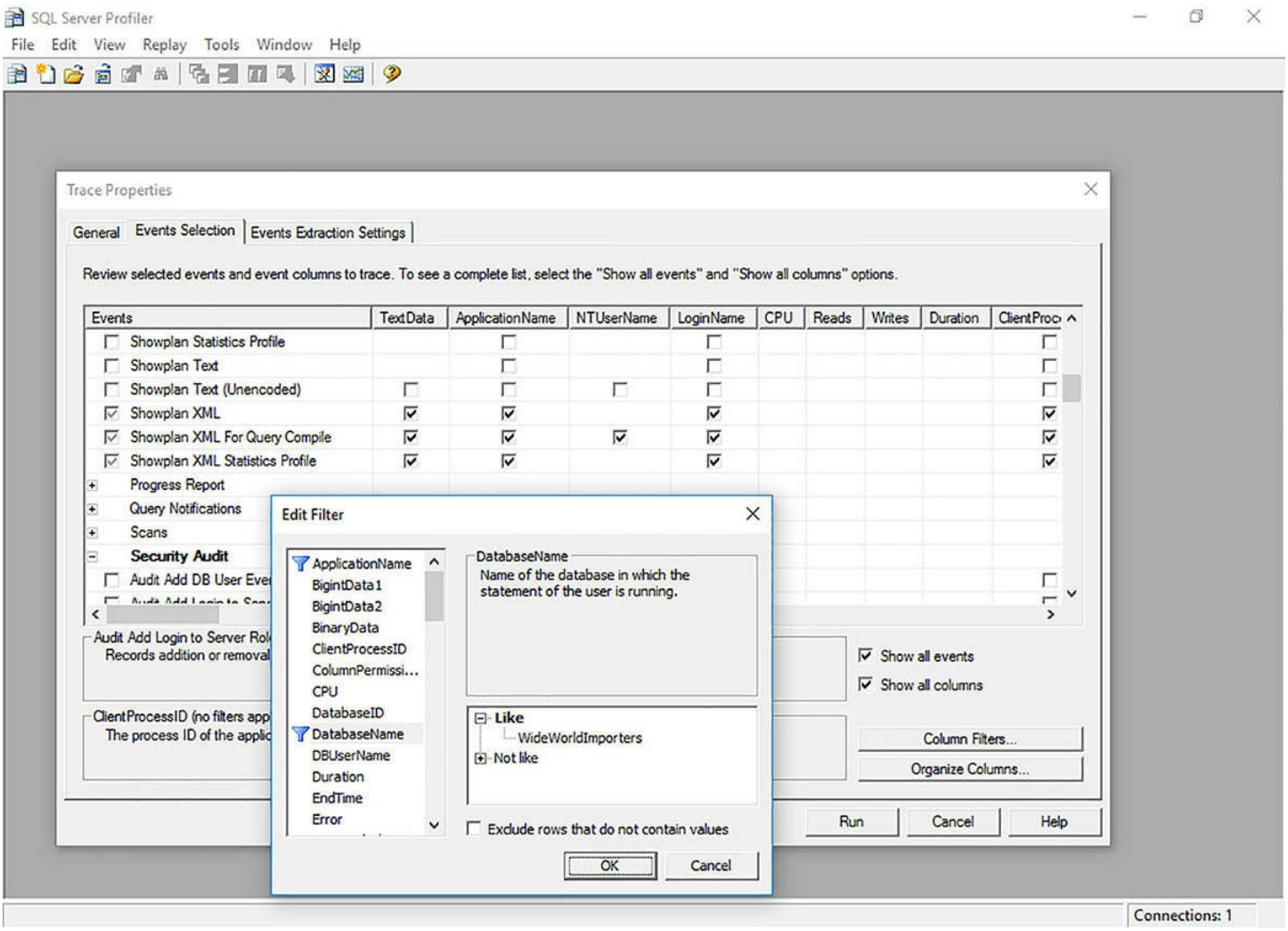


FIGURE 4-6 Adding a filter to a trace in SQL Server Profiler

After enabling a trace, you can run a query against the WideWorldImporters database, such as the one shown in Listing 4-15. After executing the query, click the Stop Selected Trace in the SQL Server Profiler toolbar. You can click on any of the events that begin with Showplan to view the graphical estimated or actual query plan, depending on the specific event you select. To save the query plan to a separate file for later review, right click the event, and then select Extract Event Data to save the file with a SQLPlan file extension. You can then open this file in SQL Server Management Studio.

Identify poorly performing query plan operators

Not only can a query plan tell you the specific steps performed by SQL Server during query execution, it can also help you discover which step in the sequence is performing poorly. Each step in the query plan is a separate operation performed by the database engine and is represented as an icon known as an *operator*. As you analyze a graphical query plan, you should check for the following conditions that can affect query performance:

- **Query plan optimization** You can find this property by right clicking the first operator in the plan (for example, SELECT) and selecting Properties. In the Properties window, look for the Reason For Early Termination Of Statement

Optimization property. If it is Good Enough Plan Found, then proceed with further analysis of the query plan. If the value is Timeout, you should spend time tuning your query because the property indicates that the current query plan is not optimal.

- **Operators** In particular, operators requiring a lot of memory (such as a *Sort*) or blocking operators can contribute to performance problems in a query. We describe potential issues with query plan operators in more detail later in this section.
- **Arrow width** The width of arrows between operators is an indicator relative to the number of rows affected by the operation. If you see one operation outputs a wide arrow, while the arrow preceding a SELECT operator is narrow, the query must process many rows before returning a small number of rows in the result set. In this case, you might investigate whether you can add a filter to the query to reduce the number of rows for the earlier operation and thereby improve overall query performance.
- **Operator cost** Each operator's contribution to the overall cost of the query is represented as a percentage value. As you analyze the query plan, look for the operators with the highest costs.
- **Warnings** When the optimizer detects a problem, it includes a warning in an operator's properties and displays a warning icon on the operator in the graphical query plan. This is a significant clue that the query performance is likely to suffer. If you see a warning, you should take steps to tune your query or optimize your environment to eliminate the warning.

Let's consider some examples of query plan operators that can adversely affect performance. Before executing the ad hoc queries in this section, click the Include Actual Execution Plan button once to enable the graphical query plan for each query.

Note Understanding query plans for queries using the WideWorldImporters database

The size of the WideWorldImporters database is too small to illustrate performance problems with the use of certain operators. Nonetheless, we use queries against this database to familiarize you with the appearance of specific poorly performing operators in a query plan, explain the conditions in which SQL Server might use them, and suggest steps you can take to improve query performance.

Exam Tip

Understanding query plan operators is important both in the real world and on the exam. Be prepared for questions that present two possible query plans and ask you to choose the more optimal of the two query plans.

Table Scan operator

As we explained in Chapter 1, SQL Server must read a heap row by row to find the rows for a query. This operation can perform slowly when run against a large table. Try this query:

Click here to view code image

```
SELECT *
FROM Warehouse.VehicleTemperatures;
```

In the Execution Plan window, as shown in Figure 4-7, you can see that SQL Server used a *Table Scan* operator that represents 100% of the query cost. Note also the size of the arrow between the SELECT and Table Scan operators to indicate a relatively large result set. In this case, the table is memory-optimized, so the performance cost of reading 659,998 rows for this query is minimal. On the other hand, if this table were not memory-optimized and you were investigating poor query performance, the presence of the Table Scan operator should lead you to consider adding a clustered index to the table or look for ways to filter the query to return fewer rows.

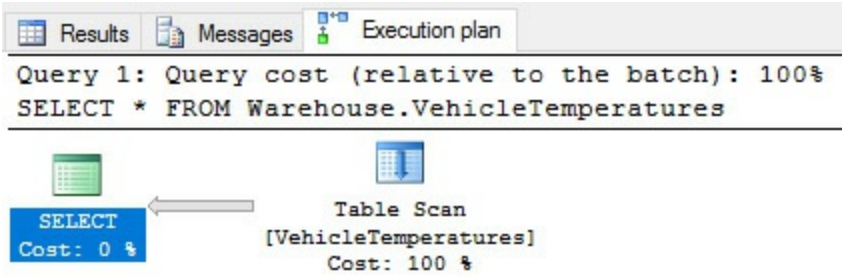


FIGURE 4-7 Table Scan operator in a query plan

Clustered Index Scan operator

Another potential problem for query performance is the use of the Clustered Index Scan operator. It is similar in concept to the Table Scan operator in that SQL Server must scan all the data. This might occur when the query must return so many rows that using the index to find specific rows is no longer advantageous, index selectivity is too low, or statistics are obsolete. To see this operator in action, execute the following query:

Click here to view code image

```
SELECT *
FROM Warehouse.StockGroups;
```

As you can see in Figure 4-8, SQL Server uses the Clustered Index Scan operator to process the query. You can also see which index is used, PK_Warehouse_StockGroups. This clustered index is relatively small (as you can see by the size of the arrows between operators), so performance is not an issue here. However, the presence of a Clustered

Index Scan operator can indicate the source of poor query performance and merits further investigation to determine whether you can modify the query by adding a WHERE clause to return only the needed rows, as long as a proper index exists for the column used in the filter. Importantly, the WHERE clause must have a predicate that includes a column on one side of the operator and an expression on the other side with both the column and expression having the same data type. If you use the LIKE operator in the WHERE clause, you cannot use a wildcard as the first character in the search string.

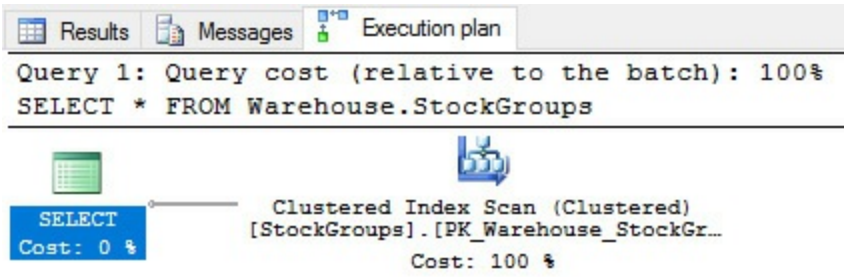


FIGURE 4-8 Clustered Index Scan operator in a query plan

That said, an index scan is not necessarily a bad operation. It is more efficient than a table scan by comparison. Nonetheless, if you need to find a way to improve query performance, your next step is to add a filter. To see what happens when you do this, execute the following query:

```
SELECT *
FROM Warehouse.StockGroups
WHERE StockGroupID = 1;
```

Now SQL Server uses a Clustered Index Seek operator, as shown in Figure 4-9. This change to the query plan is good because a seek operator can be one of the faster data retrieval methods used by SQL Server. In particular, a Clustered Index Seek is a preferred operator because all data in the table is also included in the index and SQL Server can return results directly from the index without performing additional steps.

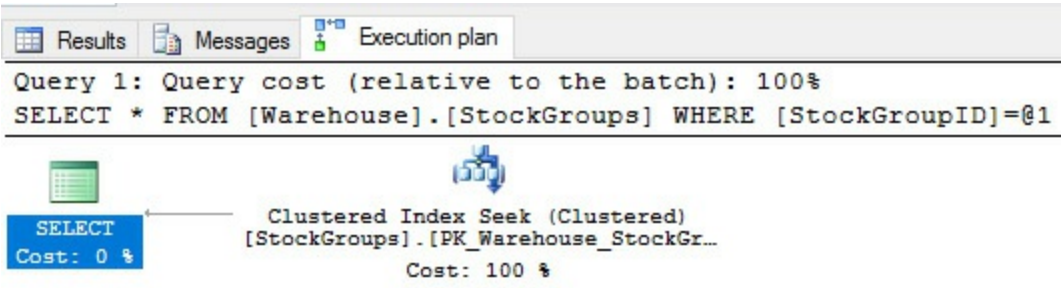


FIGURE 4-9 Clustered Index Seek operator in a query plan

Notice also that SQL Server converted the predicate to @1 instead of using the actual value of 1 that was requested in the query. That way, SQL Server can reuse the query plan when the same query executes with a different value in the WHERE clause. This reuse is known as simple parameterization.

Index Seek (NonClustered) and Key Lookup (Clustered) operators

Like the Clustered Index Seek operator, the Index Seek (NonClustered) operator is a much better operator to see in a query plan than a Table Scan or a Clustered Index Scan operator because it can selectively find rows in the index rather than read all the rows, although this behavior is dependent on the query. On the other hand, if the non-clustered index is not a covering index, the query plan also includes a Key Lookup (Clustered) operator, which adds a slight overhead to query performance. To see a query plan with these characteristics, as shown in Figure 4-10, execute this query:

Click here to view code image

```
SELECT
    StockGroupID,
    StockGroupName,
    ValidFrom,
    ValidTo
FROM Warehouse.StockGroups
WHERE StockGroupName = 'Novelty Items';
```

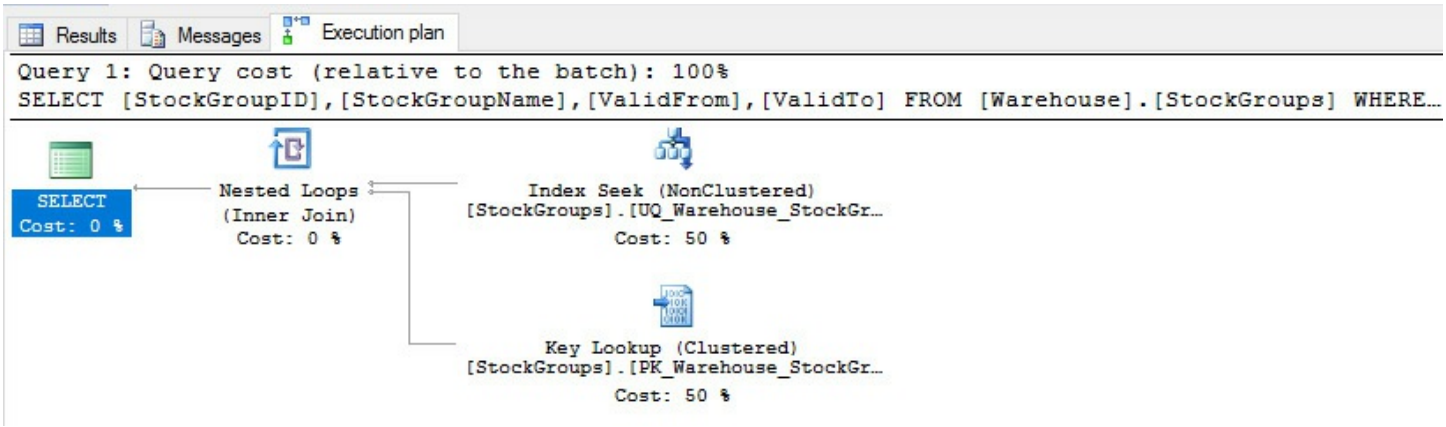


FIGURE 4-10 Index Seek (NonClustered) and Key Lookup (Clustered) operator in a query plan.

In this case, the index used by the Index Seek (NonClustered) operator contains only the StockGroupName column, but the query references other columns in the SELECT clause, so it must use the Key Lookup (Clustered) operator to get the additional columns for each row. Incidentally, if the query references columns in JOIN conditions or the WHERE clause, SQL Server includes the Key Lookup (Clustered) operator in the query plan even if those columns are not also in the SELECT clause. Therefore, whenever you see the Key Lookup (Clustered) operator in combination with an Index Seek (NonClustered) operator, consider creating a covering index by adding the necessary columns to the index key or as included columns.

Note RID Lookup operator versus Key Lookup (Clustered) operator

When a query accesses a table without a clustered index, SQL Server uses the

RID Lookup operator instead of the Key Lookup (Clustered) operator. The net effect of each operator type has a similar effect on query performance which can be mitigated by the addition of a covering index or a clustered index.

Sort operator

The Sort operator can also increase the cost of a query. Consider the query plan shown in Figure 4-11 that results from executing the following query which includes an ORDER BY clause containing a column that is not used in an index:

```
SELECT *
FROM Warehouse.StockItems
ORDER BY StockItemName;
```

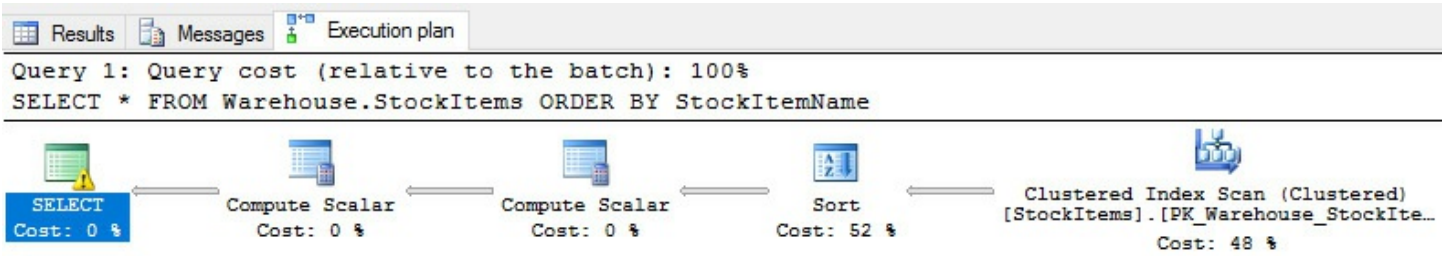


FIGURE 4-11 Sort operator in a query plan for a query sorting by a non-key column.

Contrast the query plan in Figure 4-11 with the one in Figure 4-12 that is created by executing this query:

```
SELECT *
FROM Warehouse.StockItems
ORDER BY StockItemID;
```

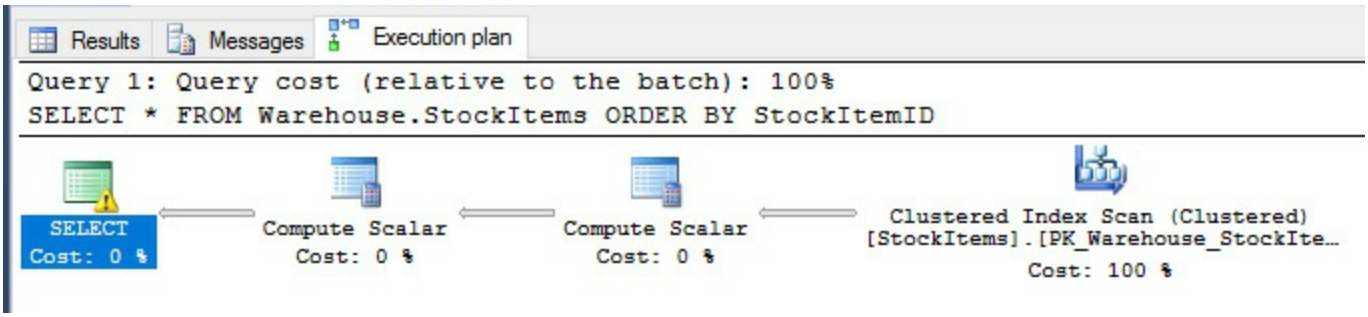


FIGURE 4-12 Sort operator in a query plan for a query sorting by a key column

Because a clustered index is already sorted, SQL Server no longer requires a Sort operator when the ORDER BY clause includes a key column from the clustered index. With this in mind, look for Sort operators with a significant query cost and consider adding the sort columns to a clustered index on the table. Another potential solution is to reduce the number of rows to sort by adding a WHERE clause to the query.

Another consideration when working with a Sort operator is the volume of data that SQL Server must sort. The query optimizer uses the estimated number of rows and the average

row size to calculate the minimum amount of memory required to perform the operation and the amount of memory needed to perform the operation entirely in memory. If the actual number of rows to be sorted is larger than the estimate due to obsolete statistics, the operation spills to tempdb. You can identify this type of memory problem in the actual query plan when you see a warning symbol on the Sort, as shown in Figure 4-13, or Hash Match operators.

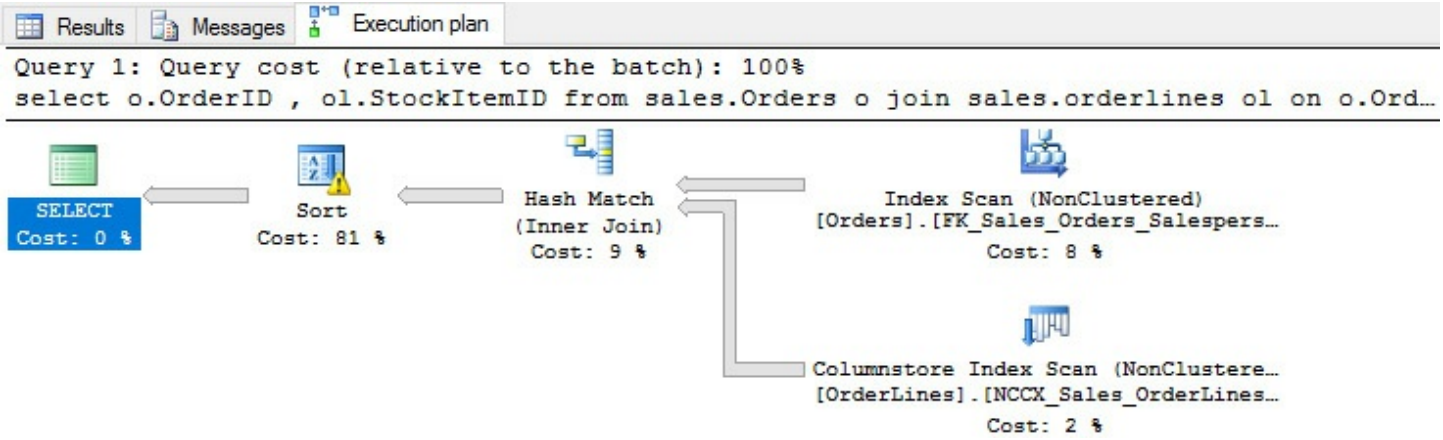


FIGURE 4-13 Sort and Hash Match operators with warning symbols in actual query plan

When you hover the cursor over the Sort operator, the tooltip includes a warning that the operator used tempdb to spill data, as shown in Figure 4-14.

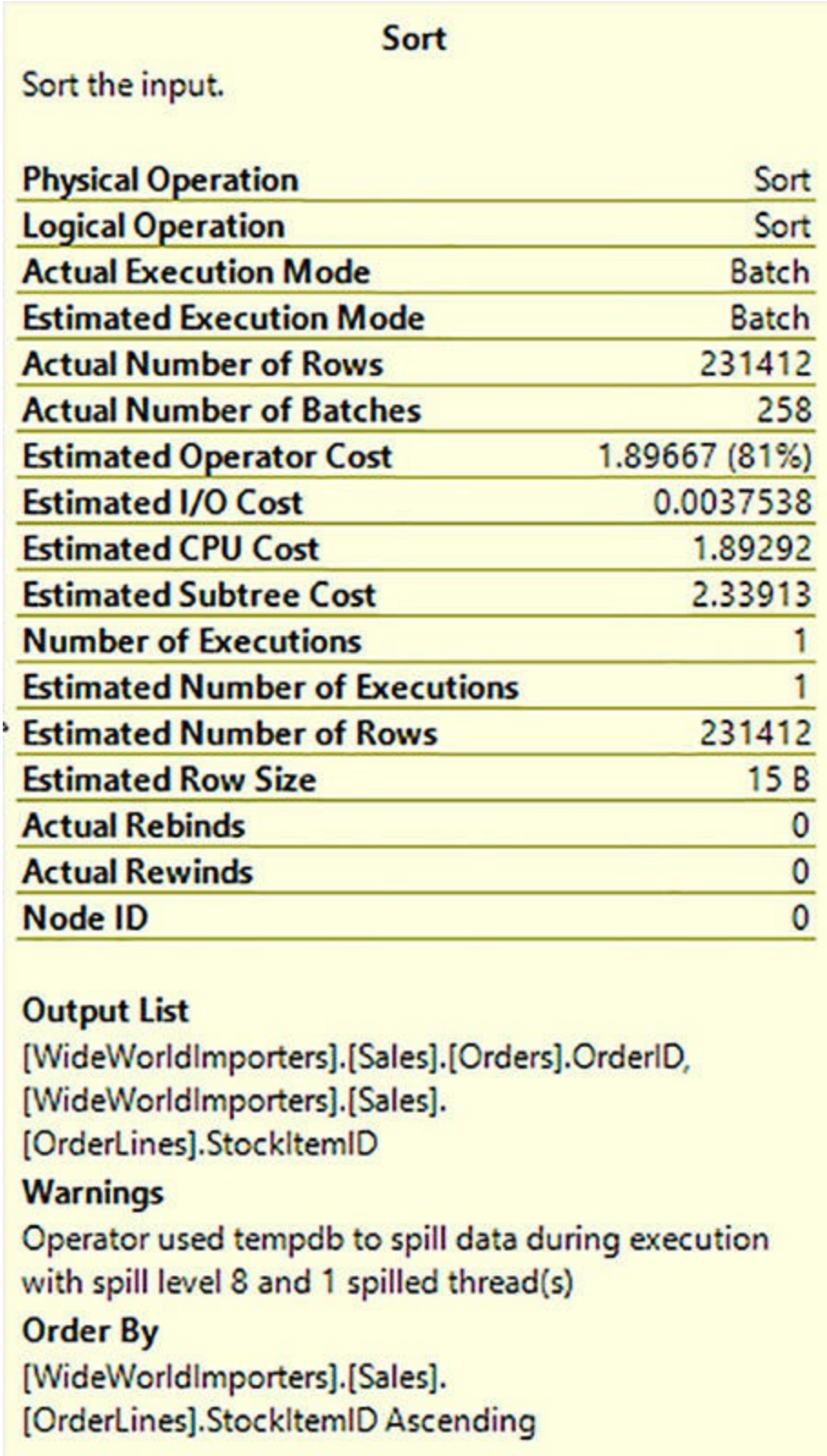


FIGURE 4-14 Sort and Hash Match operators with warning symbols in actual query plan

The query performance will suffer when the sort must use tempdb instead of memory. Use the tooltip for the SELECT operator to check the Memory Grant property which shows how much memory that SQL Server is allocating to the query. In SQL Server 2016, you can now add a query hint to request a minimum memory grant size as a percentage of the default limit to override the minimum memory per query property that is set on the server like this:

Click here to view code image

```
OPTION(min_grant_percent = 100)
```

Hash Match (Aggregate) operator

Aggregations in a query can have a negative effect on performance and should be reviewed carefully. Figure 4-15 shows the query plan created for the following aggregate query:

Click here to view code image

```
SELECT
    YEAR(InvoiceDate) AS InvoiceYear,
    COUNT(InvoiceID) AS InvoiceCount
FROM Sales.Invoices
GROUP BY YEAR(InvoiceDate);
```

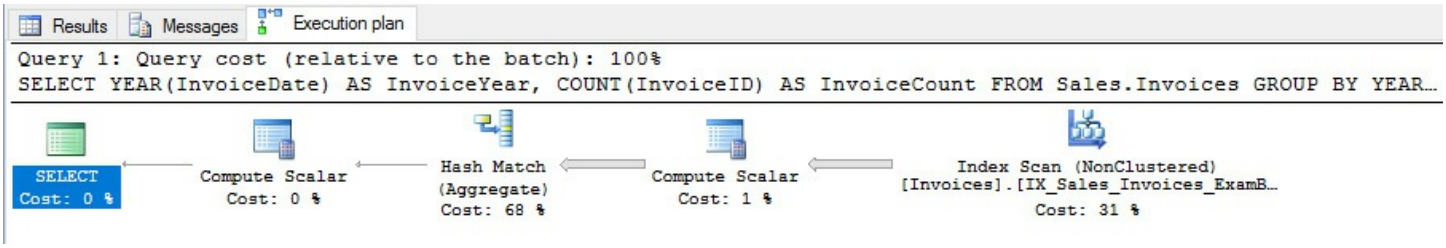


FIGURE 4-15 Hash Match (Aggregate) operator in a query plan.

In this case, the Hash Match (Aggregate) operator to group the rows from the Index Scan (NonClustered) operator contributes a significant percentage to the query cost. To perform this aggregation, SQL Server creates a temporary hash table in memory to count the rows by invoice year. Notice the larger width of the arrow sending data into the Hash Match (Aggregate) operator as compared to the width of the arrow sending the results to the next operator as an indicator that a larger row set has been reduced to a smaller row set by the operation.

Options to consider for minimizing the impact on performance when performing aggregations is to minimize the number of rows to aggregate where possible or to use an indexed view to pre-aggregate rows. Execute the statements in Listing 4-19 to set up an indexed view to improve the previous query's performance.

Click here to view code image

```
DCREATE VIEW Sales.vSalesByYear
WITH SCHEMABINDING
AS
    SELECT
    YEAR(InvoiceDate) AS InvoiceYear,
    COUNT_BIG(*) AS InvoiceCount
FROM Sales.Invoices
GROUP BY YEAR(InvoiceDate);
GO
```

```
CREATE UNIQUE CLUSTERED INDEX idx_vSalesByYear
    ON Sales.vSalesByYear
    (InvoiceYear);
GO
```

Now you can execute the earlier query again, even though it does not reference the indexed view directly, to see how the query optimizer takes advantage of the indexed view in the query plan shown in Figure 4-16.

Click here to view code image

```
SELECT
    YEAR(InvoiceDate) AS InvoiceYear,
    COUNT(InvoiceID) AS InvoiceCount
FROM Sales.Invoices
GROUP BY YEAR(InvoiceDate);
```

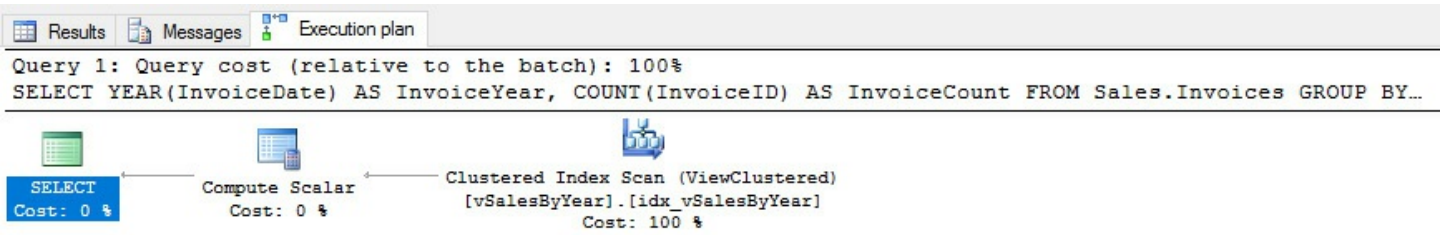


FIGURE 4-16 Query plan using an indexed view

As a result of the addition of the indexed view, SQL Server no longer requires the Hash Match (Aggregator) operator and instead uses a Clustered Index Scan (ViewClustered) operator to retrieve data. Because the data is pre-aggregated, the index scan is much faster in this case than it would be against an index containing all rows in the table.

Hash Match (Inner Join) operator

Thus far, the queries we have examined have been relatively simple and read data from only one table. Now let's consider a query that combines data from multiple tables to produce the query plan shown in Figure 4-17:

Click here to view code image

```
SELECT
    si.StockItemName,
    c.ColorName,
    s.SupplierName

FROM Warehouse.StockItems si
INNER JOIN Warehouse.Colors c ON
    c.ColorID = si.ColoriD
INNER JOIN Purchasing.Suppliers s ON
    s.SupplierID = si.SupplierID;
```

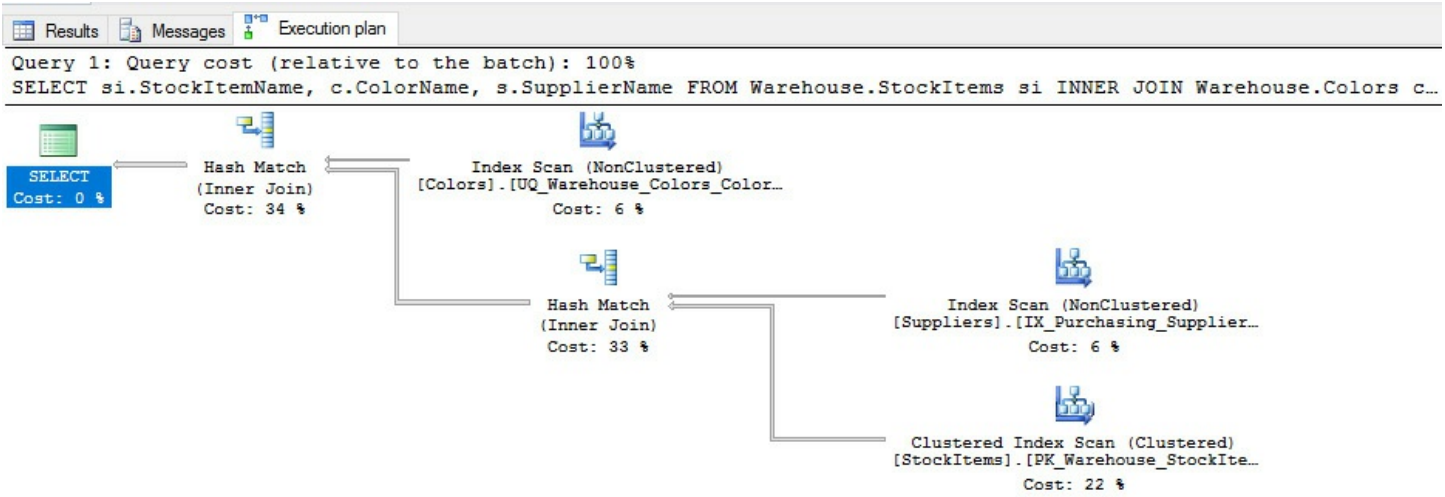


FIGURE 4-17 Hash Match (Inner Join) operator in a query plan

In this example, we see the addition of the Hash Match (Inner Join) operator in two places in the query plan. We also see that these two operations have the two highest costs in the plan and therefore should be the first operations we assess for possible optimization. SQL Server uses this operator when it puts data into temporary hash tables so that it can match rows in two different data sets and produce a single result set. Specifically, SQL Server converts, or *hashes*, rows from the smaller data set into a value that is more efficient for comparisons and then stores these values in a hash table in tempdb. Then it compares each row in the larger data set with the hash table to find matching rows to join. As long as the smaller data set is in fact small, this comparison operation is fast, but performance can suffer when both data sets are large. Furthermore, if a query requires many of these operations, tempdb might experience memory pressure. Last, it's important to note that the Hash Match (Inner Join) operator is a blocking operator as it requires SQL Server to gather data from each data set before it can perform the join.

In Figure 4-17, the Hash Match (Inner Join) operator combines the results of the Index Scan (NonClustered) and the Clustered Index Scan operators that gets SupplierID and SupplierName by scanning a non-clustered index on the Suppliers table and StockItemName, SupplierID, and ColorID by scanning the clustered index on the StockItems table. This result set becomes input for the second usage of the operation and is combined with ColorName set from an index scan on the Colors table.

Your options for improving the query performance based on this query plan include adding or revising indexes, filtering the data by using a WHERE clause, or fixing a WHERE clause that prevents the query optimizer from using an existing index. Let's try adding indexes to the Suppliers and StockItems tables, as shown in Listing 4-20. The index on the Suppliers table includes the SupplierID column used for the JOIN operation and the SupplierName column to return in the final query results. Similarly, the index on the StockItems table includes the ColorID and SupplierID columns used for JOIN operations and the StockItemName column to return in the final query results.

LISTING 4-20 Add indexes to eliminate Hash Match (Inner Join) operators

Click here to view code image

```
CREATE NONCLUSTERED INDEX
IX_Purchasing_Suppliers_ExamBook762Ch4_SupplierID
    ON Purchasing.Suppliers
(
    SupplierID ASC,
    SupplierName
);
GO
CREATE NONCLUSTERED INDEX
IX_Warehouse_StockItems_ExamBook762Ch4_ColorID
    ON Warehouse.StockItems
(
    ColorID ASC,
    SupplierID ASC,
    StockItemName ASC
);
```

After adding the indexes, execute the following query to see the new query plan, as shown in Figure 4-18:

Click here to view code image

```
   SELECT
    si.StockItemName,
    c.ColorName,
    s.SupplierName
FROM Warehouse.StockItems si
INNER JOIN Warehouse.Colors c ON
    c.ColorID = si.ColoriD
INNER JOIN Purchasing.Suppliers s ON
    s.SupplierID = si.SupplierID;
```

After adding the indexes, execute the following query to see the new query plan, as shown in Figure 4-18:

Click here to view code image

```
SELECT
    si.StockItemName,
    c.ColorName,
    s.SupplierName

FROM Warehouse.StockItems si
INNER JOIN Warehouse.Colors c ON
    c.ColorID = si.ColoriD
INNER JOIN Purchasing.Suppliers s ON
```

```
s.SupplierID = si.SupplierID;
```

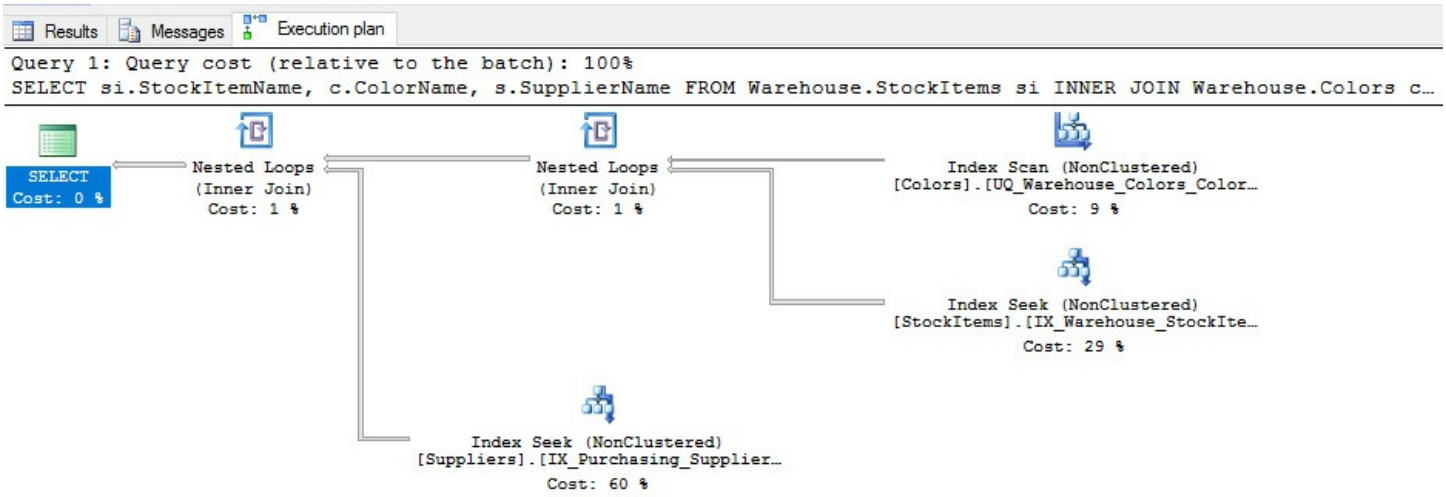


FIGURE 4-18 Query plan after adding usable indexes to eliminate Hash Match (Inner Join) operators

In the new query plan, SQL Server replaces the Hash Match (Inner Join) operators with Nested Loops operators and replaces two of the index scan operations with Index Seek (NonClustered) operators which should significantly improve performance even when large tables are queried. The Nested Loops operator is an efficient operation that compares two data sets row by row. For each row output by the top operator in the query plan (known as the inner data set), which is the index scan on the small Colors table, SQL Server scans the rows output by the bottom operator (known as the outer data set), which is the index seek on the StockItems table. Then the output of this operation becomes the inner data set for the second Nested Loops operator in the query plan. For each row in this new inner data set, SQL Server scans the output from the index seek on the Suppliers table. Notice that the cost of the Nested Loops operators in the new query plan is significantly lower than the cost of the Hash Match (Inner Join) operators shown in Figure 4-17. Create efficient query plans using Query Store

The query plan selected by the query optimizer is not guaranteed to be the most efficient plan. When working with a version earlier than SQL Server 2016, you can capture query plans from the procedure cache periodically, but run the risk of losing access to query plans when the server comes under memory pressure and begins evicting older query plans from the cache. With Query Store in SQL Server 2016 and Azure SQL Database, you can capture and analyze information about all query plans generated over time for a query. You can then force SQL Server to use the query plan that you determine to be most efficient based on the criteria that best meets your business requirements.

Query Store properties

By default, Query Store is not enabled. You can enable it at the database level in SQL Server Management Studio by using Object Explorer to navigate to the database for which you want to enable Query Store. Right-click the database name to open the Database

Properties dialog box and click the Query Store tab. In the Operation Mode (Requested) drop-down list, select Read Write.

> **Note Changing the Query Store behavior**
>
> You can later change the Operation Mode (Requested) property to Read Only when you want to retain existing query plans and execution statistics information available in the query store, but no longer want to add new information. SQL Server automatically switches to this mode when the query store reaches its maximum allocated space. To disable Query Store, change this property to Off.

As shown in Figure 4-19, there are several other properties that you can configure to manage the query store:

- **Data Flush Interval (Minutes)** The frequency in minutes at which SQL Server writes data collected by the query store to disk.
- **Statistics Collection Interval** The granularity of time for which SQL Server aggregates runtime execution statistics for the query store. You can choose one of the following intervals: 1 minute, 5 minutes, 10 minutes, 15 minutes, 30 minutes, 1 hour, or 1 day. If you capture data at a high frequency, bear in mind that the query store requires more space to store more finely grained data.
- **Max Size (MB)** The maximum amount of space allocated to the query store. The default value is 100 MB per database. If your database is active, this value might not be large enough to store query plans and related information.
- **Query Store Capture Mode** The specification of the types of queries for which SQL Server captures data for the query store. You can choose one of the following options:
  - **None** The query store stops collecting data for new queries, but continues capturing data for existing queries.
  - **All** The query store captures data for all queries.
  - **Auto** The query store captures data for relevant queries. It ignores infrequent queries and queries with insignificant compile and execution duration.
- **Size Based Cleanup Mode** The specification of whether the cleanup process activates when the query store data approaches its maximum size (Auto) or never runs (OFF).
- **Stale Query Threshold (Days)** The number of days that SQL Server keeps data in the query store.

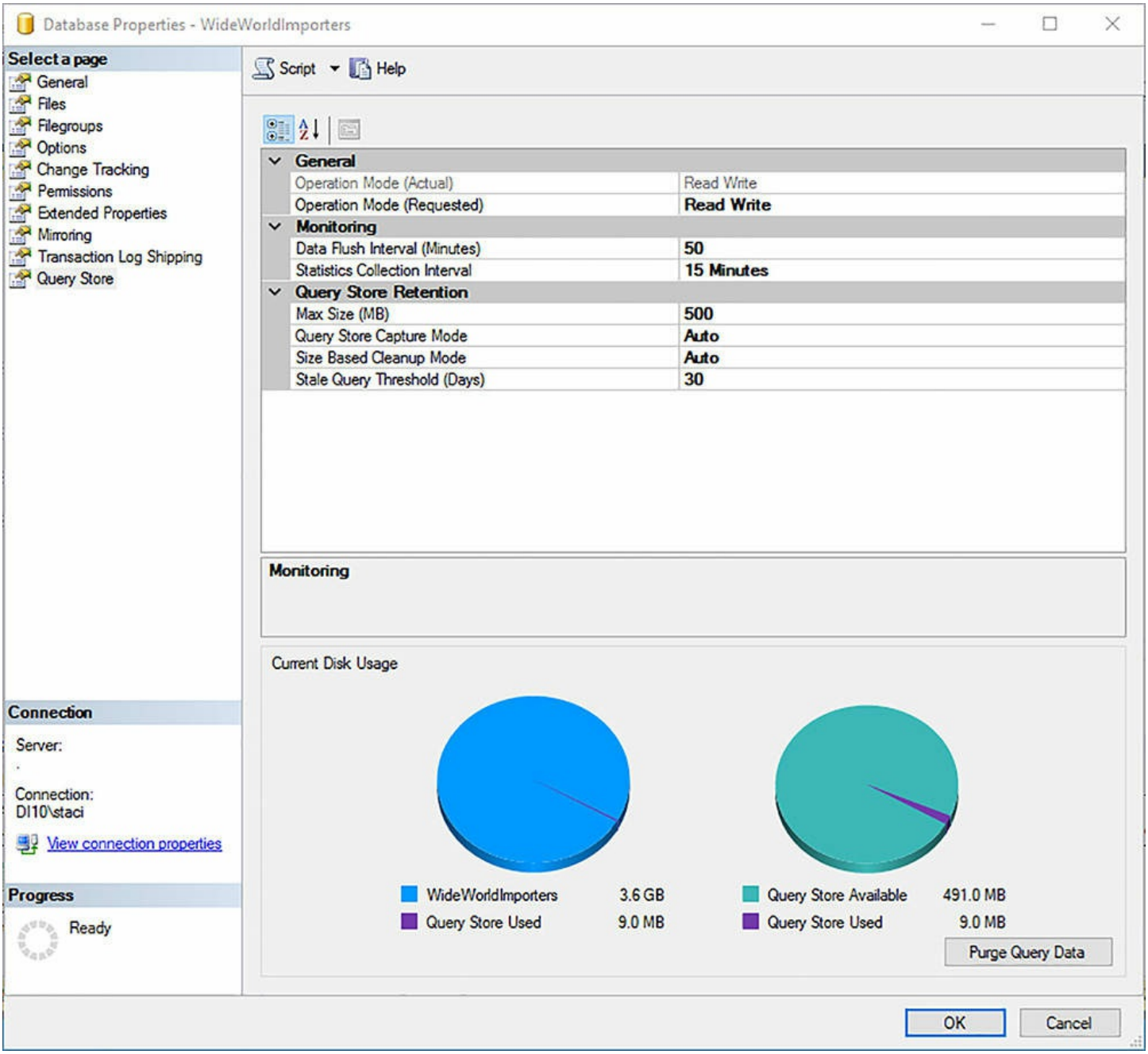


FIGURE 4-19 Query Store properties configurable in the Database Properties dialog box

> **Note Query store enabled by default for WideWorldImporters database**
>
> The query store for WideWorldImporters database is already enabled when you restore it. The current disk usage that displays for you will differ from that shown in Figure 4-19 as these metrics depend on the volume of query activity that has occurred on your computer to date.

You can also use the statement shown in Listing 4-21 to enable the query store, replacing <databasename> with the name of the database that you want to configure.

LISTING 4-21 Enable the query store for a database and set its properties

Click here to view code image

```
ALTER DATABASE <databasename>
    SET QUERY_STORE = ON
    (
        OPERATION_MODE = READ_WRITE ,
        CLEANUP_POLICY = ( STALE_QUERY_THRESHOLD_DAYS = 30 ),
        DATA_FLUSH_INTERVAL_SECONDS = 3000,
        MAX_STORAGE_SIZE_MB = 500,
        INTERVAL_LENGTH_MINUTES = 50
    );
```

You can clear the data from the query store by clicking Purge Query Data on the Query Store tab of the Database Properties dialog box or by executing either of the statements shown in Listing 4-22.

LISTING 4-22 Purge data from the query store

Click here to view code image

```
--Option 1: Use the ALTER DATABASE statement
ALTER DATABASE <databasename>
SET QUERY_STORE CLEAR ALL;
GO

--Option 2: Use a system stored procedure
EXEC sys.sp_query_store_flush_db;
GO
```

Query Store components

The query store captures information about query plans and runtime execution statistics until the maximum space allocation is reached. You can review this data in the following DMVs:

- **sys.query_store_plan** Query plan information, such as Showplan XML, the number of compilations, the date and time of the initial and last compilations, the last execution date and time, and the average and most recent duration of compilation, among other details. The query plan available in this DMV is the estimated plan only.
- **sys.query_store_query** Aggregated runtime execution statistics for a query, including CPU binding, memory, optimization, and compilation statistics. This information is stored at the statement level and not at the batch level which is different from the behavior of sys.dm_exec_query_stats.
- **sys.query_store_query_text** The text of the executed query.

- **sys.query_store_runtime_stats** Runtime execution statistics for a query, such as first and last execution date and time, the number of executions, statistics (average, last, minimum, maximum, and standard deviation) for query duration, CPU time, logical IO reads and writes, physical IO reads and writes, CLR time, DOP, maximum used memory, and row counts.
- **sys.query_store_runtime_stats_interval** The start and end times defining the intervals during which SQL Server collects runtime execution statistics for the query store.

As an example, you can query the DMVs to find the top query with the highest average logical reads and its corresponding query plan, as shown in Listing 4-23.

LISTING 4-23 Top 5 queries with highest average logical reads

Click here to view code image

```
USE WideWorldImporters;
GO
SELECT TOP 1
    qt.query_sql_text,
    CAST(query_plan AS XML) AS QueryPlan,
    rs.avg_logical_io_reads
FROM sys.query_store_plan qp
INNER JOIN sys.query_store_query q
  ON qp.query_id = q.query_id
INNER JOIN sys.query_store_query_text qt
    ON q.query_text_id = qt.query_text_id
INNER JOIN sys.query_store_runtime_stats rs
    ON qp.plan_id = rs.plan_id
ORDER BY rs.avg_logical_io_reads DESC;
```

You can use the following system stored procedures to manage the query store:

- **sp_query_store_flush_db** Flush the portion of the query store currently in memory to disk. This stored procedure takes no arguments.
- **sp_query_store_force_plan** Force SQL Server to use a specified query plan for a specified query. You provide identifiers for the query and plan as arguments for this stored procedure.
- **sp_query_store_remove_plan** Remove a specified query plan from the query store.
- **sp_query_store_remove_query** Remove a specified query from the query store, in addition to the query plans and runtime execution statistics related to it.
- **sp_query_store_reset_exec_stats** Reset the runtime execution statistics for a specified plan.

- **sp_query_store_unforce_plan** Keep a specified query plan in the query store, but no longer force SQL Server to use it for a specified query.

> **Need More Review? More query store monitoring examples available online**
>
> See the "Key Usage Scenarios" section of the "Monitoring Performance By Using the Query Store" article at *https://msdn.microsoft.com/en-US/library/dn817826.aspx*. Here you will find several examples of using these DMVs and system stored procedures, such as showing the last n queries executed on a database, the number of executions per query, queries having the longest average execution time in the last hour, among others.

Query Store views

An easy way to review the information available in the query store is to use Query Store views in SQL Server Management Studio. After enabling query store for a database, a Query Store node appears below the database node and contains four views, as shown in Figure 4-20.

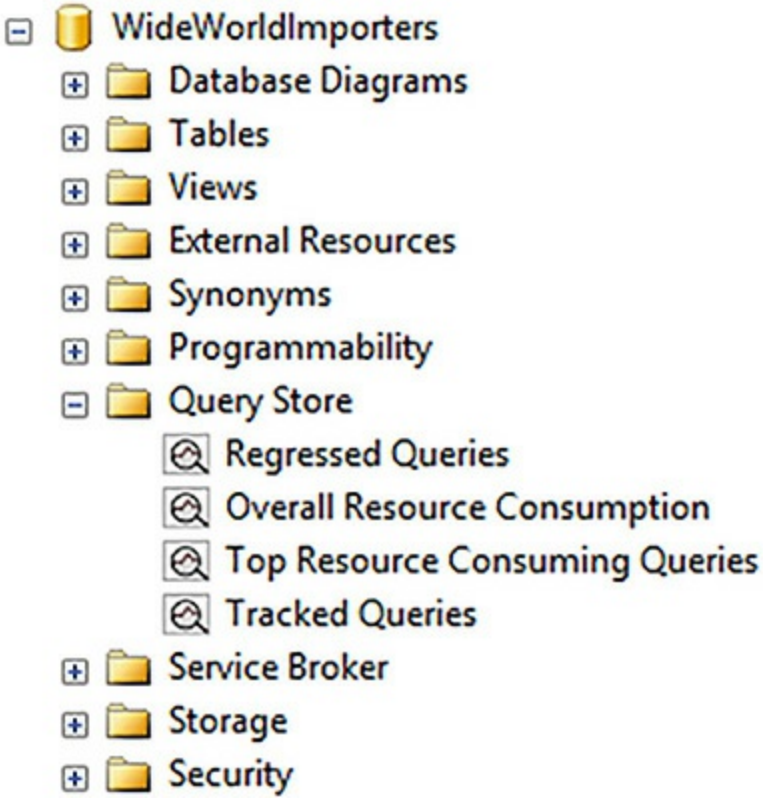


FIGURE 4-20 Query Store views for a database in SQL Server Management Studio's Object Explorer

Before we the review contents of these views, let's execute the statements in Listing 4-24 to create a test environment in which a new database and table is added. The table is populated with a random 9,999 rows of random values with an ID of 1 and one row with

an ID of 2 to create a skewed distribution. A primary key clustered index and a non-clustered index on the ID column are added. Next, a parameterized stored procedure to select rows from the table is added to the database. Query Store Is enabled on the database to capture query plan changes over time using an interval length of 1 minute to capture statistics at the most granular level available for better visibility of the query examples in this section. Last, the stored procedure is called with the ID associated with a large number of rows in the table. This stored procedure is called multiple times to accumulate more statistics than other background queries in the database so that you can see it more easily in the view later.

LISTING 4-24 Create test environment for Query Store

Click here to view code image

```
CREATE DATABASE ExamBook762Ch4_QueryStore;
GO
USE ExamBook762Ch4_QueryStore;
GO
CREATE SCHEMA Examples;
GO
CREATE TABLE Examples.SimpleTable(
    Ident INT IDENTITY,
    ID INT,
    Value INT);
WITH IDs
    AS (SELECT
                TOP (9999)
                ROW_NUMBER() OVER (ORDER BY (SELECT 1)) AS n
            FROM master.sys.All_Columns ac1
            CROSS JOIN master.sys.All_Columns ac2
        )
INSERT  INTO Examples.SimpleTable(ID, Value)
SELECT
    1,
    n
FROM    IDs;
GO
INSERT Examples.SimpleTable (ID, Value)
    VALUES (2, 100);
ALTER TABLE Examples.SimpleTable
    ADD  CONSTRAINT [PK_SimpleTable_Ident]
PRIMARY KEY CLUSTERED (Ident);
CREATE NONCLUSTERED INDEX ix_SimpleTable_ID
    ON Examples.SimpleTable(ID);
GO
CREATE PROCEDURE Examples.GetValues
```

```
    @PARAMETER1 INT
AS
    SELECT
        ID,
        Value
    FROM Examples.SimpleTable
    WHERE
        ID = @PARAMETER1;
GO

ALTER DATABASE ExamBook762Ch4_QueryStore
SET QUERY_STORE = ON (
    INTERVAL_LENGTH_MINUTES = 1
);

EXEC Examples.GetValues 1;
GO 20
```

At this point, the Top Resource Consuming Queries view, shown in Figure 4-21, is the only one that contains information. The default configuration includes a column chart in the top left that displays total duration by query id, a point chart in the top right that displays duration for query plans associated with the selected query (known as a *plan summary*), and a query plan for the plan identifier currently selected in the point chart.

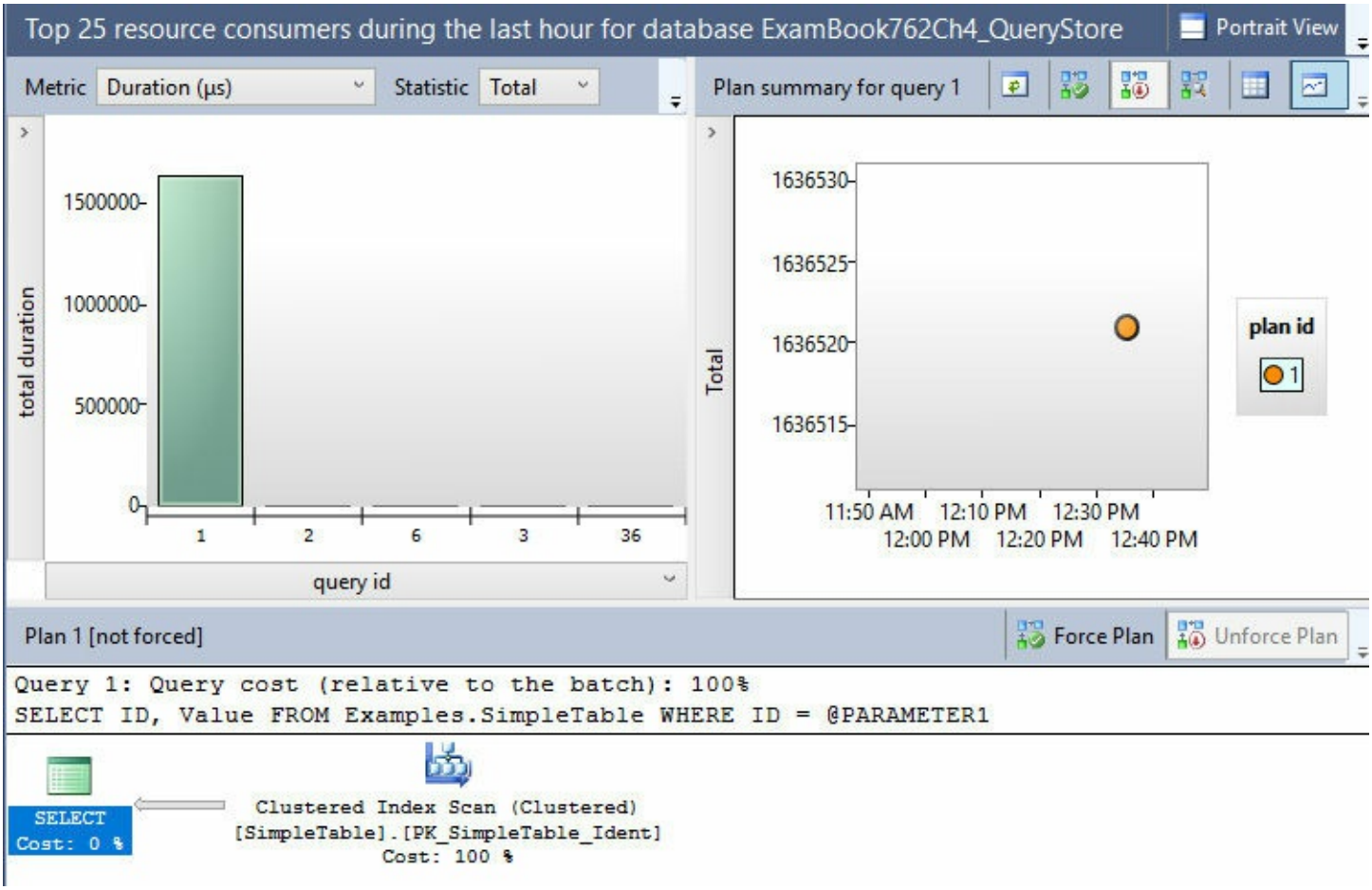


FIGURE 4-21 Top Resource Consuming Queries view

Note Working with multiple queries in the Top Resource Consuming Queries view

Depending on the activity occurring in the database, the query that you want to analyze might not be the first column in the chart as it is in Figure 4-21. You can click on each column to view the query plan associated with a query and its query text, or click the View Top Resource Consuming Queries In A Grid Format With Additional Details button in the chart's toolbar to locate your query by its query text. If you select a query in this grid and toggle back to the chart, the view retains your selection and shows the plan summary and a query plan.

Taking a closer look at the column chart, notice that each column represents a query. When multiple queries exist in the query store, the columns are sorted in descending order by the selected metric, which is currently Duration. When you hover the cursor over a column in the chart, a tooltip displays the query id, the metric and its selected statistic, and the query text. If the query includes a WHERE clause, the query text is parameterized. You can find similar information by using the sys.query_store_query, sys.query_store_query_text, and sys.query_store_runtime_stats DMVs. By using the Query Store view in SQL Server Management Studio, you can quickly visualize and access key metrics about your queries without writing any code.

The plan summary chart displays one point per query plan for the query selected in the column chart. When you hover the cursor over a point on this chart, as shown in Figure 4-22, a tooltip displays some of the information related to the selected metric that is available in the sys.query_store_runtime_stats DMV. Last, notice the graphical query plan shows that SQL Server used a Clustered Index Scan operator to retrieve rows for the SELECT statement in the stored procedure. An index scan is used instead of a seek because the number of rows for the parameter value of 1 is high relative to the size of the table.

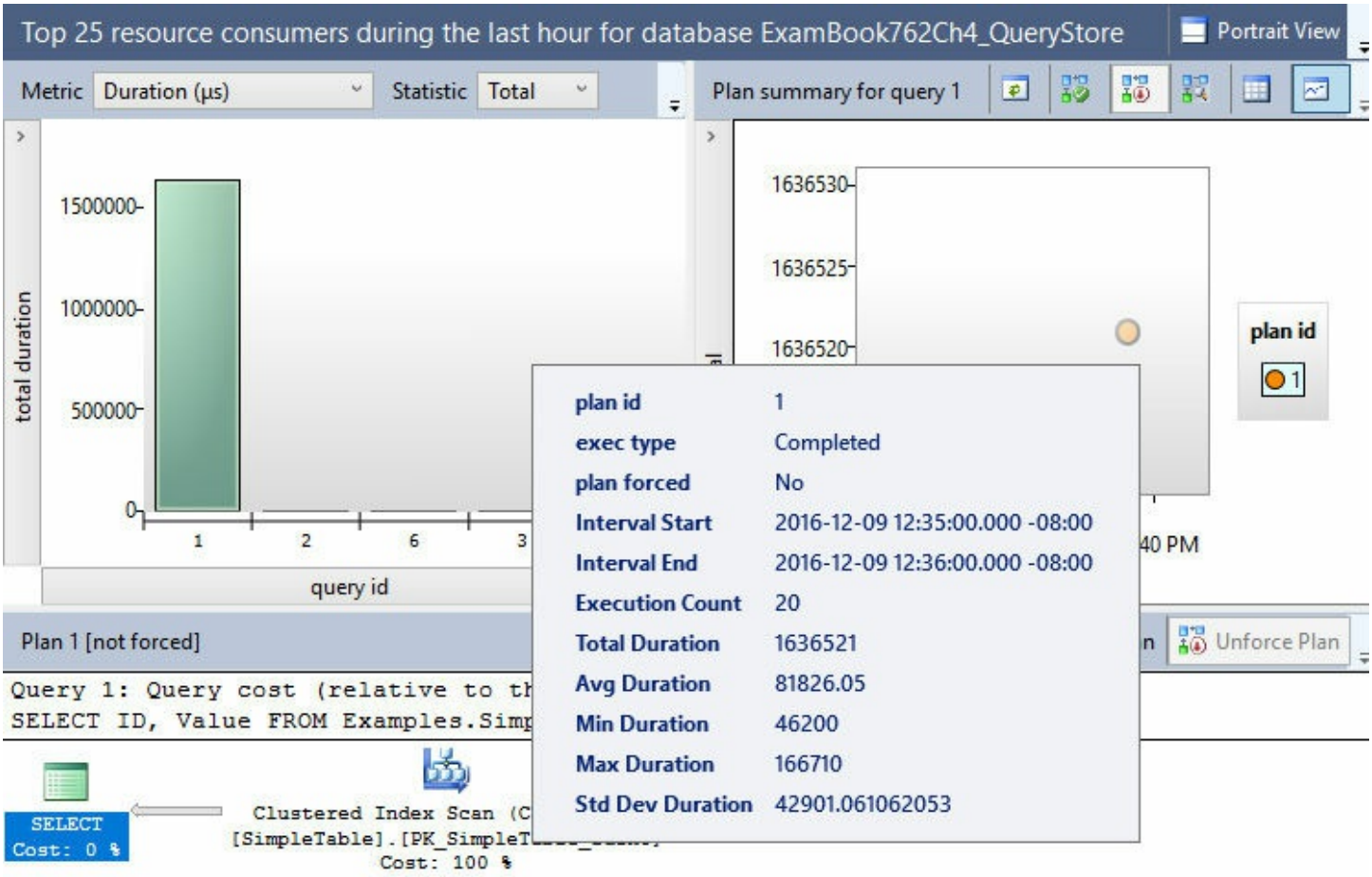


FIGURE 4-22 Plan summary for a selected query and its associated runtime statistics

You can change the configuration of the charts by changing the y-axis of the column chart and by selecting new metric and statistic values in the respective drop-down lists above the column chart. For example, select Logical Reads in the Metric drop-down list, and Avg in the Statistic drop down list, and point to your query's column in the chart to view the corresponding metric value, execution count, and number of query plans for the selected query.

Now let's modify the query slightly by changing the parameter to the other possible ID value in the table, and execute it by using the code shown in Listing 4-25.

LISTING 4-25 Execute stored procedure with new parameter value

```
EXEC Examples.GetValues 2;
GO
```

Return to the Top Resource Consuming Queries dashboard and click the Refresh button above the column chart to update it. When you click the query's column in the column chart, the plan summary chart now has two points that are associated with the same query plan, as shown in Figure 4-23. That means the index scan for the query executed to get rows with ID 1 was also used to get the one row with ID 2. This is condition is a result of *parameter*

sniffing in which the query optimizer uses the estimated rows from the first execution of the stored procedure to select a query plan and then uses it for all subsequent executions without considering the estimated rows for the new parameter values. In the current example, an index seek is a more efficient operator for retrieving the one row for ID 2, but it is not considered due to the parameter sniffing behavior.

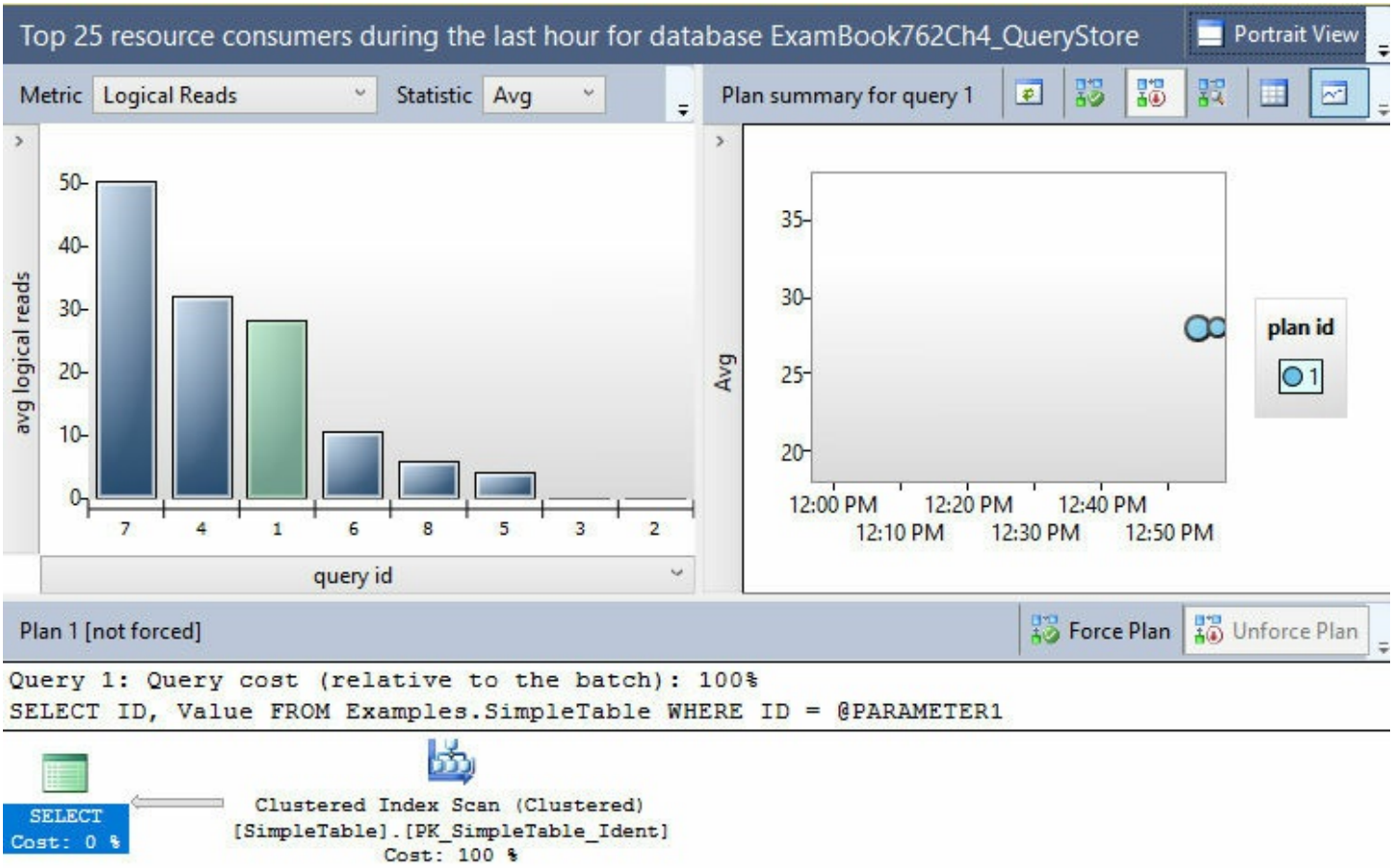


FIGURE 4-23 Plan summary with two points for the same Plan ID

Let's clear the procedure cache and see what happens when SQL Server must generate a new query plan rather than use the one created previously. To do this, execute the statements shown in Listing 4-26. Click the Refresh button in the Query Store view to see the effect of the last query execution. When you click the Plan ID with the higher identifier in the legend, you can see its new query plan, as shown in Figure 4-24. Notice the disparity between the points in the point chart which indicates the new query plan has a lower average logical read value than the first query plan and is therefore more efficient. As you can see in the graphical query plan at the bottom of the view, SQL Server used an index seek operation to retrieve a single row from the table.

LISTING 4-26 Execute stored procedure after clearing procedure cache

```
DBCC FREEPROCCACHE();
GO
```

```
EXEC Examples.GetValues 2;
GO
```

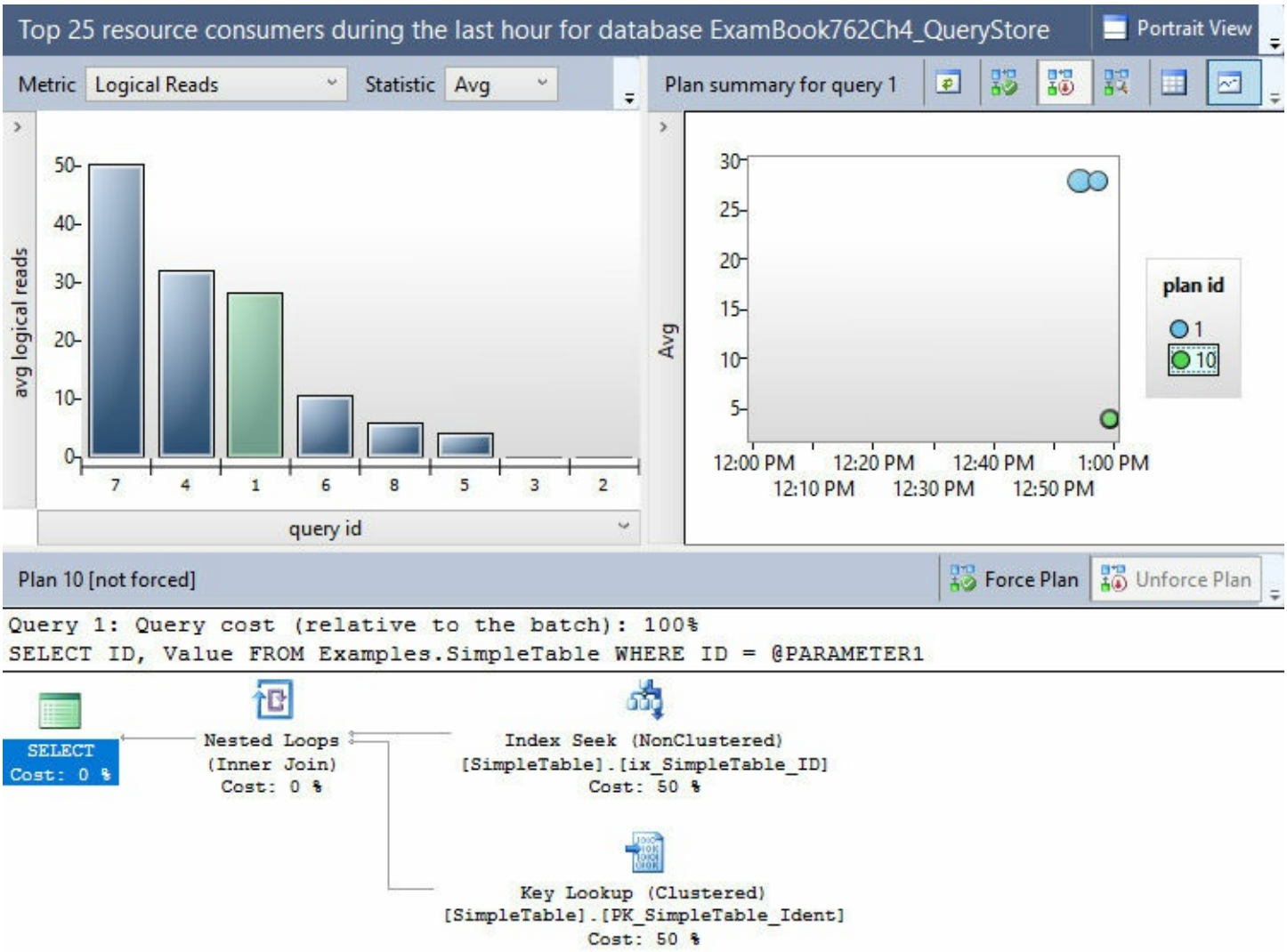
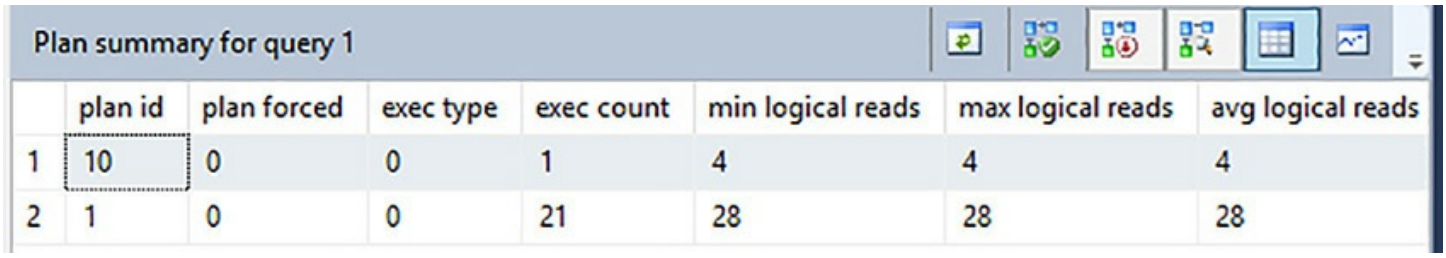


FIGURE 4-24 An improved query plan in the Plan Summary after clearing the procedure cache

Another way to compare the differences between the query plans is to view their respective metrics. To do this, click the View Plan Summary In A Grid Format button in the toolbar to switch the plan summary from a chart to a grid, as shown in Figure 4-25.



	plan id	plan forced	exec type	exec count	min logical reads	max logical reads	avg logical reads
1	10	0	0	1	4	4	4
2	1	0	0	21	28	28	28

FIGURE 4-25 Plan Summary metrics in a grid format

When you click a row in the grid, you can see the associated graphical query plan, but you can only view one graphical query plan at a time this way. If you want to compare

query plans, click the View Plan Summary In A Chart Format button in the toolbar to switch back to the point chart, click one plan identifier in the legend and then, while holding the Shift key, click the other plan identifier. Next, click the Compare The Plans For The Selected Query In A Separate Window button in the toolbar. In the Showplan Comparison window that opens, shown in Figure 4-26, you can more easily compare the differences between the two plans. The top query plan (which was the last to execute) uses an index seek to retrieve one row, whereas the bottom query plan uses an index scan to retrieve many rows and performs less efficiently for small rowsets.

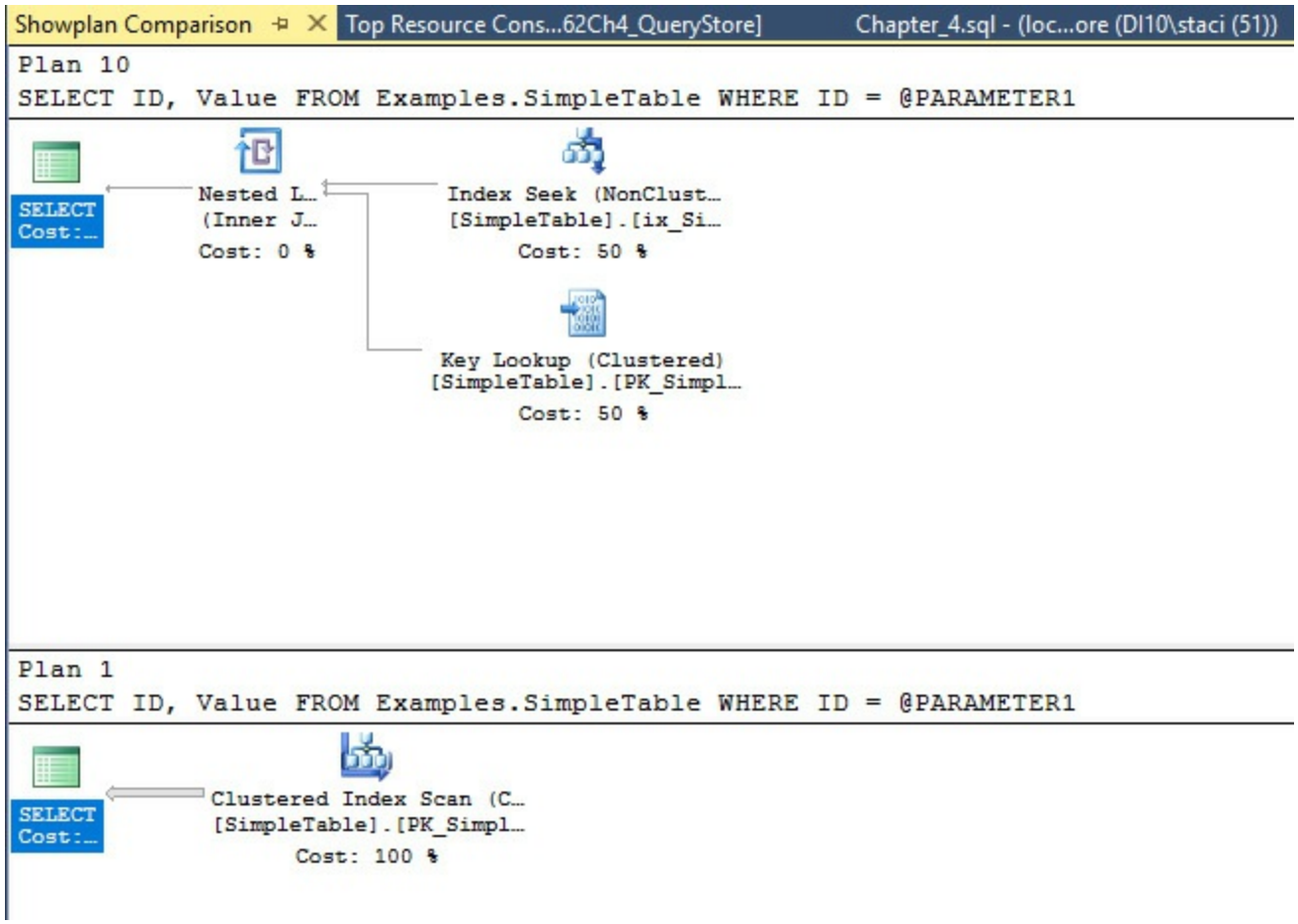


FIGURE 4-26 Showplan Comparison between query plans captured by Query Store

Let's say that the normal query pattern for this particular stored procedure is to retrieve a limited number of rows. In that case, the use of the query plan that uses the index seek is preferable. An advantage of using Query Store is the ability to force this plan to be used for all executions of the same query. Returning to the Top Resource Consuming Queries view, click the plan identifier for the last query plan (with the lower average logical reads) and then click the Force Plan button to require SQL Server to use this query plan for all future queries. You must confirm that you want to force the selected plan before SQL Server accepts the change.

Run the query shown in Listing 4-27 to retrieve most of the rows from the table, and then refresh the Top Resource Consuming Queries view to check the results, as shown in Figure 4-27. Although there is a forced plan for the query that should be reused, the Plan Summary

chart now shows a third plan identifier, Plan ID 15, for which the average logical reads is significantly higher than it was for Plan ID 10, the forced plan. (You can identify the forced plan id by the check mark on the point in the chart.)

LISTING 4-27 Execute stored procedure with new parameter value and forced query plan

```
EXEC Examples.GetValues 1;
GO
```

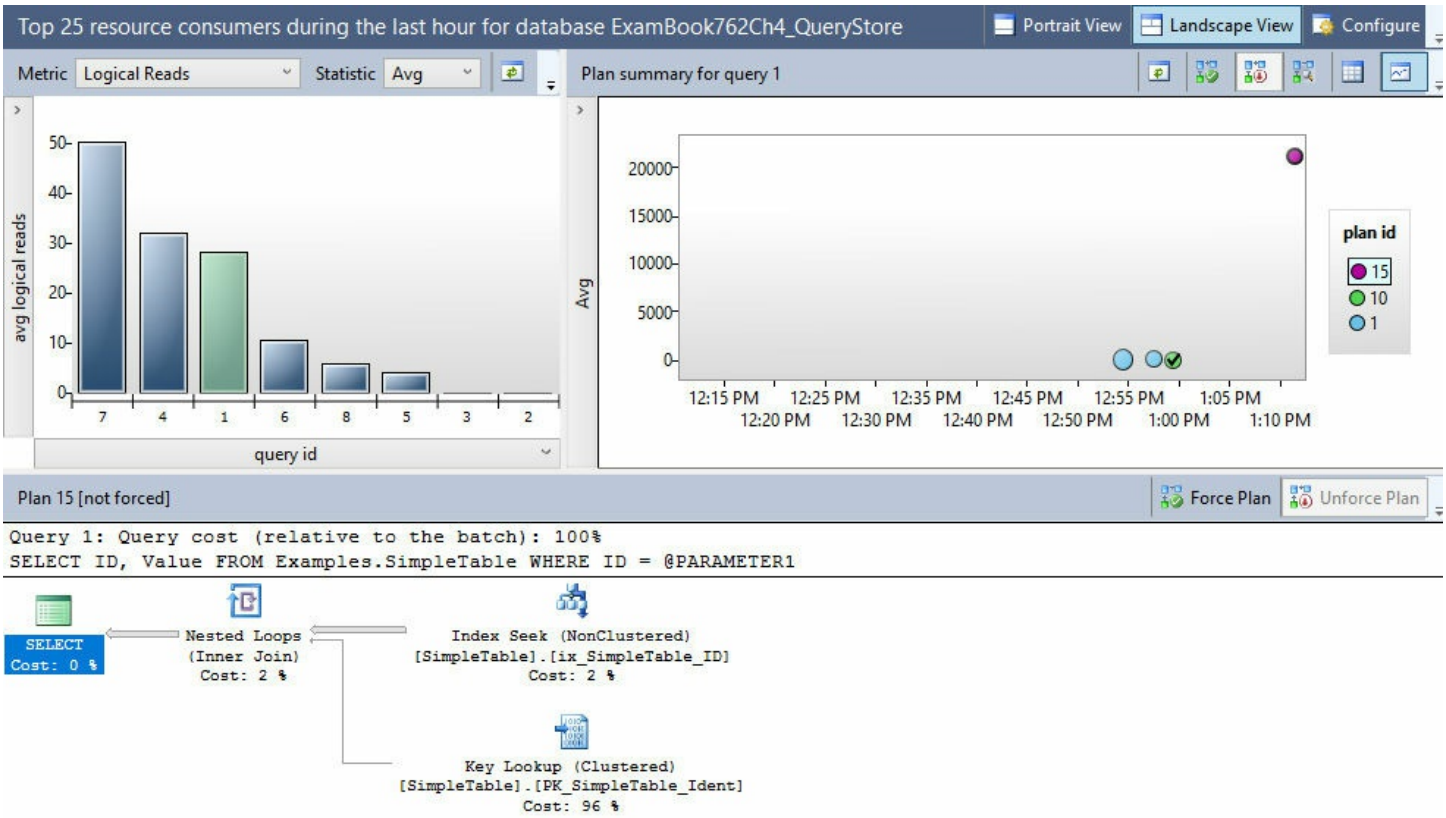


FIGURE 4-27 A new query plan in the Query Store view based on the forced plan

If you check the graphical query plan for Plan ID 10 and Plan ID 15, you find that both query plans use an index seek operation to retrieve rows. SQL Server generated a new query plan for the stored procedure due to the difference in estimated rows, but the same operations are used. Although this is not the most efficient operation when retrieving a relatively high number of rows, as we discussed previously, it might be a reasonable trade-off when the majority of executions retrieves a small number of rows. If you execute the stored procedure again using a parameter value of 2 and refresh the view, another point associated with Plan ID 10 appears on the Plan Summary chart with a lower value for average logical reads.

Of course, this approach is not a suitable solution for all performance problems. Furthermore, forcing a query plan does not guarantee that SQL Server always uses that query plan. As an example, if a forced query plan is dependent on an index that is

subsequently dropped, the plan forcing will fail. For this reason, you should periodically review the status of forced plans and reasons for failure by running the query shown in Listing 4-28. You can also use the query_store_plan_forcing_failed Extended Event to monitor failed plan forcing.

LISTING 4-28 Check status of forced plans

Click here to view code image

```
SELECT
    p.plan_id,
    p.query_id,
    q.object_id,
    force_failure_count,
    last_force_failure_reason_desc
FROM sys.query_store_plan AS p
INNER JOIN sys.query_store_query AS q
    ON p.query_id = q.query_id
WHERE is_forced_plan = 1;
```

The following three Query Store views behave much like the view we explored in detail in this section, but focus on different types of queries:

- **Regressed Queries** Displays a column chart of metrics by query, a plan summary, and query plan for *regressed queries* executed in the previous hour. A regressed query is one for which a new query plan generated for a particular query is less optimal than a query plan that was previously used for the same query. Regression can happen due to changes in statistics, structural changes to the data, addition or removal or indexes, and so on. Use this view to find regressed queries and determine whether to force an earlier query plan.
- **Overall Resource Consumption** Displays overall resource consumption during the last month in separate charts: duration, execution count, CPU time, and logical reads. You can toggle between the chart view and grid view. This view does not provide access to the query plan details.
- **Tracked Queries** Displays tracked queries. You add a query to this dashboard view by selecting it in the metric chart and then clicking the Track The Selected Query In A New Tracked Queries Window. That way, you can focus on the metrics for a single query rather than try to find it among the changing set of Query IDs on the Top Resource Consuming Queries view.

Compare estimated and actual query plans and related metadata

After the query optimizer generates a query plan or uses a query plan existing in the plan

cache, the storage engine is responsible for executing the query according to that plan. However, the plan that it starts with is an *estimated query plan*. That is, it is the plan determined by the query optimizer to be the most efficient query plan based on the calculations performed by the query optimizer. At runtime, SQL Server reports an *actual query plan* to add runtime information to the estimated query plan. When you compare an estimated and actual query plan for the same query, you can see differences when either of the following situations occur:

- **Inaccurate statistics** As data is inserted into or deleted from a table, both the indexes and the distribution of data in each column change. The automatic statistics update uses a data sample rather than the entire table to reduce the overhead of the process, Therefore, statistics can become less accurate over time.
- **Structural changes** Changing the schema of a table or changing its structure also affects indexes and data distribution and causes a recompilation as does changing or dropping an index used by the query or updating statistics.

A common reason to work with estimated query plans is to evaluate performance of a query in development, particularly when the query execution time is long or when restoring the database to its state prior to query execution is challenging. You can add or change indexes or modify the query structure and then analyze changes to the estimated query plan after making these changes.

To set up an environment to compare estimated and actual query plans that differ, execute the code in Listing 4-29.

LISTING 4-29 Create test environment for comparing estimated and actual query plans

Click here to view code image

```
CREATE DATABASE ExamBook762Ch4_QueryPlans;
GO
USE ExamBook762Ch4_QueryPlans;
GO
CREATE SCHEMA Examples;
GO
CREATE TABLE Examples.OrderLines (
    OrderLineID int NOT NULL,
    OrderID int NOT NULL,
    StockItemID int NOT NULL,
    Description nvarchar(100) NOT NULL,
    PackageTypeID int NOT NULL,
    Quantity int NOT NULL,
    UnitPrice decimal(18, 2) NULL,
    TaxRate decimal(18, 3) NOT NULL,
    PickedQuantity int NOT NULL,
    PickingCompletedWhen datetime2(7) NULL,
```

```
    LastEditedBy int NOT NULL,
    LastEditedWhen datetime2(7) NOT NULL);
GO
INSERT INTO Examples.OrderLines
SELECT *
FROM WideWorldImporters.Sales.OrderLines;
GO
CREATE INDEX ix_OrderLines_StockItemID
ON Examples.OrderLines (StockItemID);
GO
```

Next, execute the code in Listing 4-30 to generate an estimated query plan. The inclusion of the SET SHOWPLAN_XML ON statement instructs SQL Server to generate the estimated plan without executing the query. As an alternative, you can use the following statements:

- **SET SHOWPLAN_TEXT ON** Returns a single column containing a hierarchical tree that describes the operations and includes the physical operator and optionally the logical operator.
- **SET SHOWPLAN_ALL ON** Returns the same information as SET SHOWPLAN_TEXT except the information is spread across a set of columns in which you can more easily see property values for each operator.

LISTING 4-30 Generate estimated query plan

Click here to view code image

```
SET SHOWPLAN_XML ON;
GO
BEGIN TRANSACTION;
    UPDATE Examples.OrderLines
        SET StockItemID = 300
        WHERE StockItemID < 100;
    SELECT
        OrderID,
        Description,
        UnitPrice
    FROM Examples.OrderLines
    WHERE StockItemID = 300;
ROLLBACK TRANSACTION;
GO
SET SHOWPLAN_XML OFF;
GO
```

Click the result row to view the graphical query plan in its own window, as shown in

Figure 4-28. Query 3 shows the use of an index seek operation in the SELECT statement.

> **Note SQL Server 2016 Service Pack 1 (SP1) behavior**
>
> If you are using SQL Server 2016 SP1 and have not applied any subsequent cumulative updates or service packs, the query plan is displayed as XML text instead of the graphical query plan.

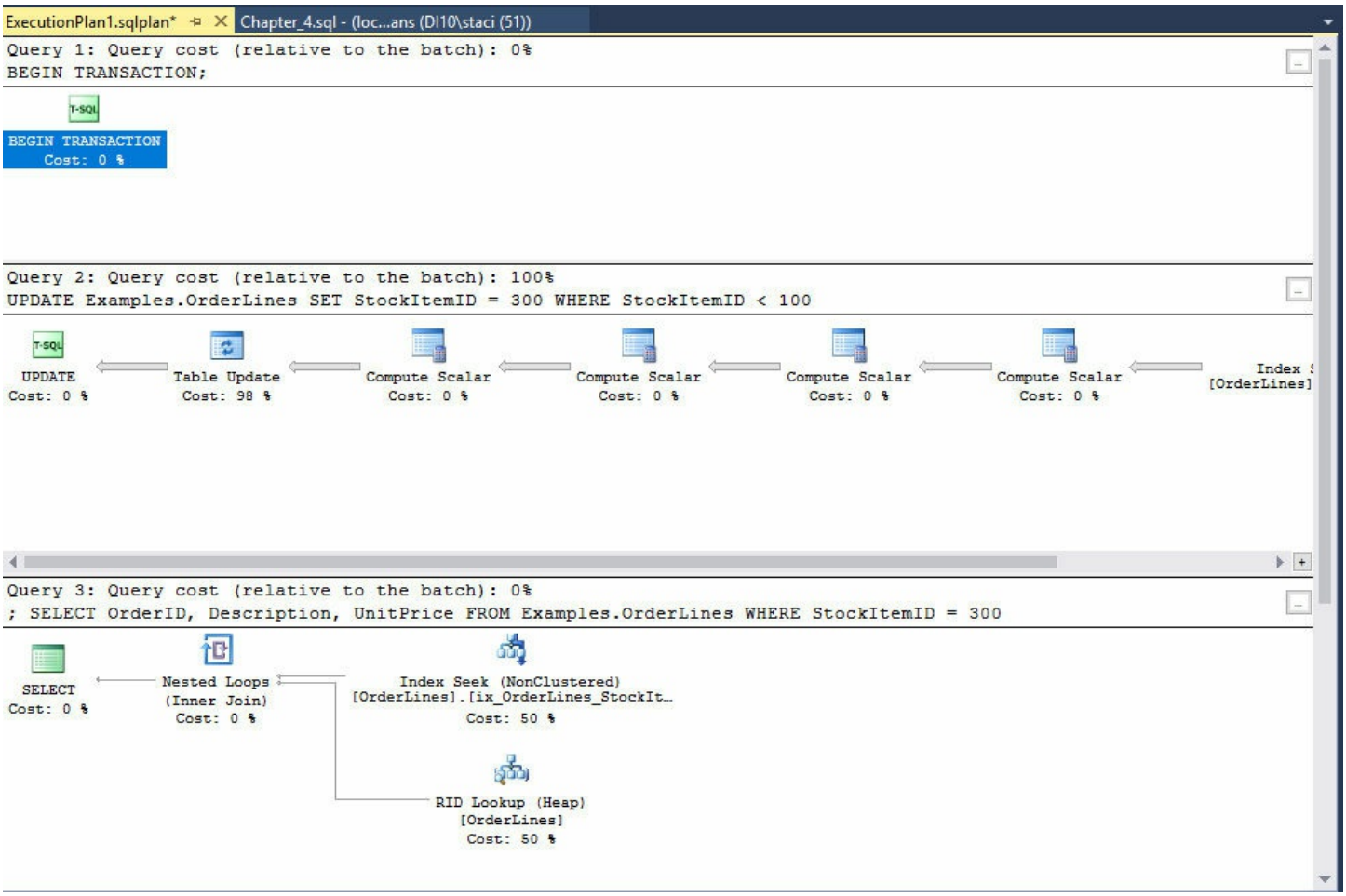


FIGURE 4-28 Estimated query plan

Let's take a closer look at the SELECT statement operations. Because this is an estimated plan, the UPDATE operation has not yet occurred. Therefore, there are no rows with StockItemID equal to 300 in the statistics. By hovering the cursor over the Index Seek (NonClustered) operator, you can display the tooltip that shows the Estimated Number of Rows is 1, as shown in Figure 4-29. It does not yet factor in the effect of the UPDATE operation. Consequently, the query optimizer selects an index seek an operation because it is efficient for a single row. However, if you check the Estimated Row Size for the UPDATE operator for Query 2 in the estimated query plan, the value is 104391, which is a significant discrepancy from the estimate for the index seek operation in Query 3.

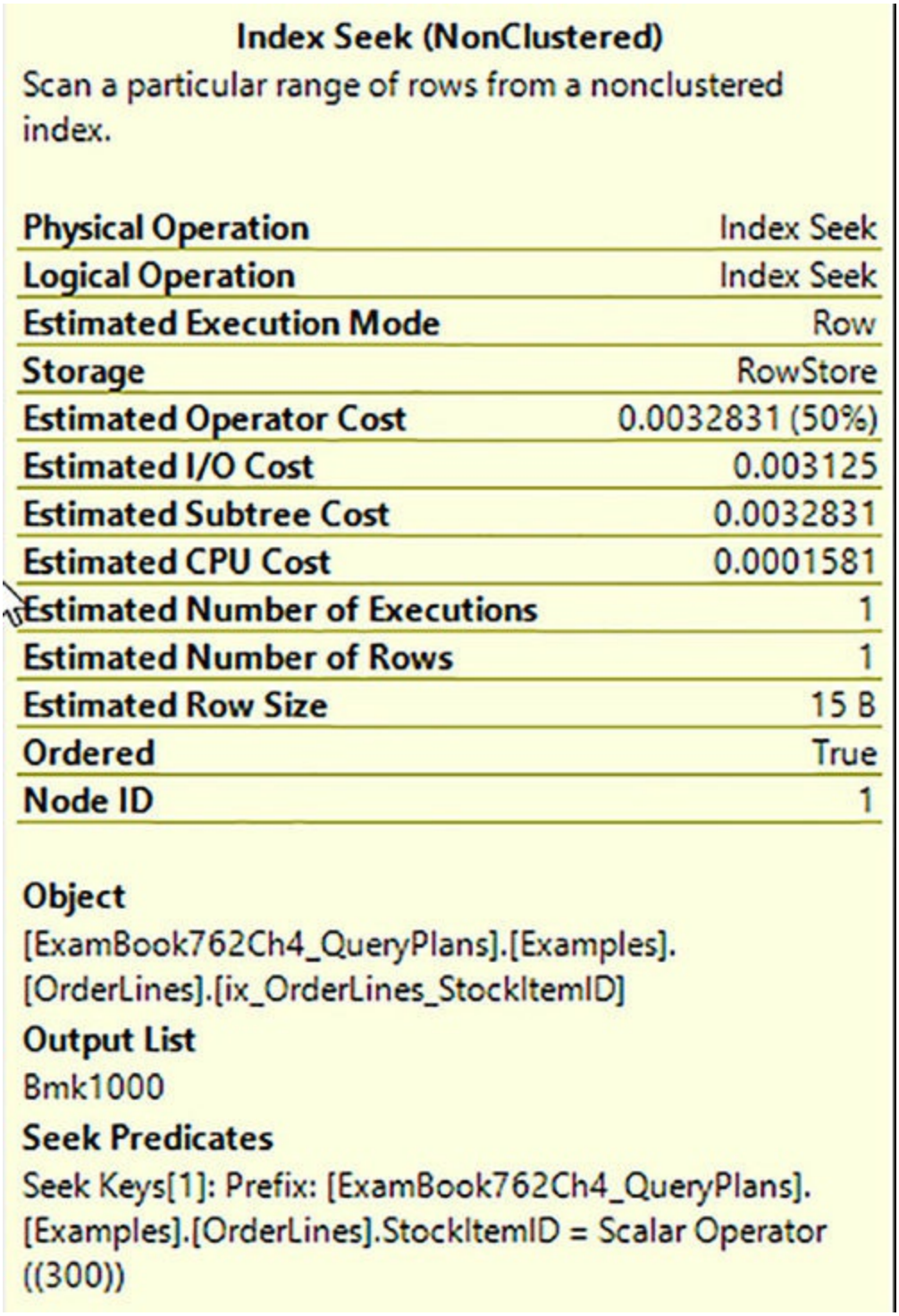


FIGURE 4-29 Estimated Row Size in estimated query plan for Index Seek (NonClustered) operator

To generate an actual query plan, execute the statements in Listing 4-31, which uses the SET STATISTICS XML ON statement to have SQL Server generate a graphical actual query plan. As an alternative, you can use the SET STATISTICS PROFILE ON statement to get the query plan information in a hierarchical tree with profile information available across columns in the result set. Figure 4-30 shows that SQL Server recognized the change of greater than 20% in the table's statistics and performed an automatic update which in turn forced a recompilation of the SELECT statement's query plan. This time the query

optimizer chose a Table Scan operator because the number of rows to retrieve is nearly half the number of rows in the table.

> **Note SQL Server 2016 SP1 behavior**
>
> If you are using SQL Server 2016 SP1 and have not applied any subsequent cumulative updates or service packs, the query plan is displayed as XML text instead of the graphical query plan.

LISTING 4-31 Generate actual query plan

Click here to view code image

```
SET STATISTICS XML ON;
GO
BEGIN TRANSACTION;
    UPDATE Examples.OrderLines
        SET StockItemID = 300
        WHERE StockItemID < 100;
    SELECT
        OrderID,
        Description,
        UnitPrice
    FROM Examples.OrderLines
    WHERE StockItemID = 300;
ROLLBACK TRANSACTION;
GO
SET STATISTICS XML OFF;
GO
```

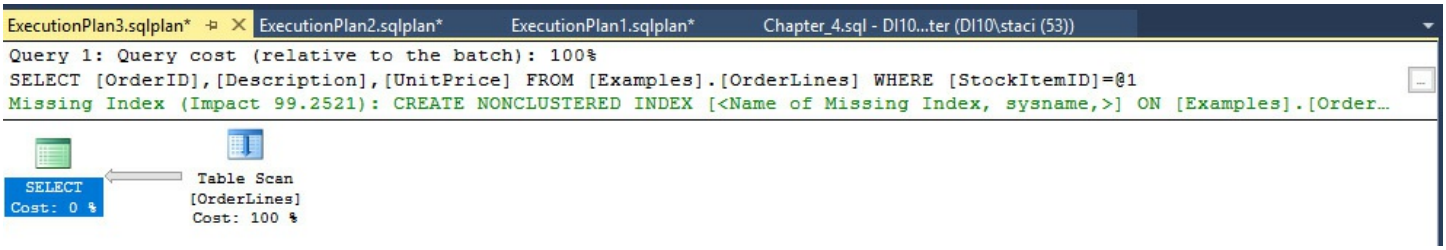


FIGURE 4-30 Actual query plan

Checking the tooltip for the Table Scan operation, shown in Figure 4-31, notice the Estimated Number of Rows is 104066 to reflect the updated statistics and the Actual Number of Rows is 104391.

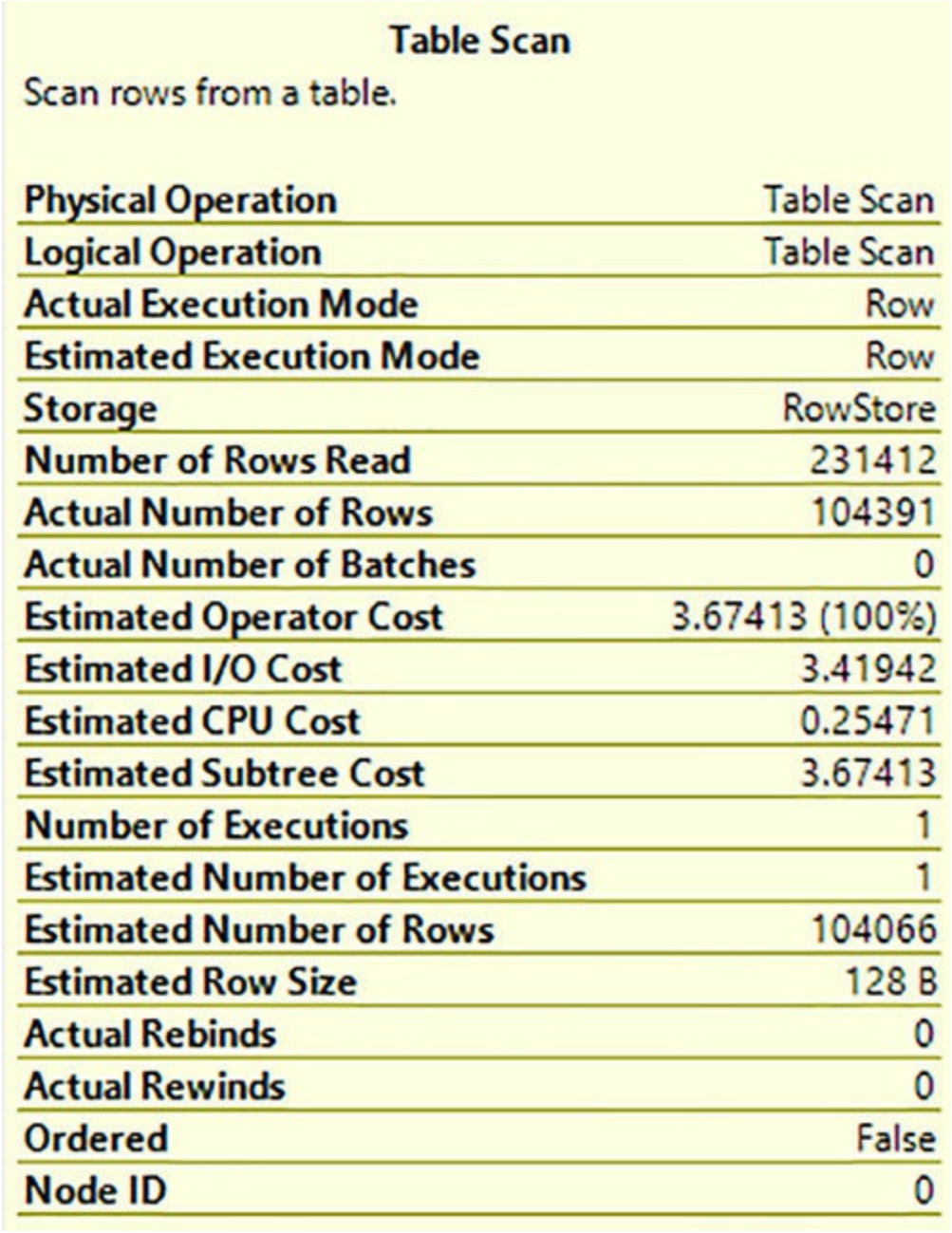


FIGURE 4-31 Estimated Number of Rows and Actual Number of Rows in actual query plan for Table Scan operator.

Need More Review? Additional resource for query plans

For a deep dive into query plans, also known as execution plans, download Grant Fritchey's *SQL Server Execution Plans, Second Edition*, a free e-book available at *https://www.simple-talk.com/books/sql-books/sql-server-execution-plans-second-edition-by-grant-fritchey/*.

Configure Azure SQL Database Performance Insight

Query Performance Insight is the name of a feature available in Azure SQL Database that allows you to review the effect of queries on database resources, identify long-running

queries, or create custom settings to review query workloads. This feature available only with Azure SQL Database V12 and requires you to enable Query Store on your database before you can analyze queries. Because Query Store is enabled by default for a V12 database, you do not need to perform this extra step unless you previously disabled it.

Note Creating a SQL Database in the Azure portal

To work with the monitoring tools in SQL Database, you must have an Azure account and description. You must then create a SQL Database and associate it with a new or existing server. Last, you must configure the firewall settings to enable your IP address to access the database.

If you do not currently have an account, you can set up a free trial at *https://azure.microsoft.com/en-us/free/*. Then connect to the Azure portal at *https://portal.azure.com*. Next, to create a new sample database, click SQL Databases in the navigation pane on the left side of the screen, and then click Add to open the SQL Database blade. Here you provide a name for your database, select a subscription, select Create New in the Resource Group section, and provide a name for the resource group. In the Select Source dropdown list, select Sample, and then in the Select Sample dropdown list, select AdventureWorksLT [V12]. Click Server, click Create A New Server, provide a server name, a server admin login, password and password confirmation, and location. Be sure to keep the default selection of Yes for Create V12 Server (Lastest Update) as Query Performance Insight works only with SQL Database V12. Click Select to create the server. In the SQL Database blade, click Pricing Tier, select the Basic tier, and then click the Select button. For the sample database, you can use the lowest service tier level to minimize charges associated with this service. When you no longer need to work with the database, be sure to delete it in the Azure portal to avoid incurring ongoing charges. In the SQL Database blade, click Create to finalize the creation of the sample database. When the database is ready, it appears in the list of SQL Databases. You might need to click Refresh several times to see it appear.

When the SQL Database is available, click the dataset to open its blade, and then click the server name to open the server's blade. Click Show Firewall Settings, click Add Client IP, and then click Save to enable your connection to the SQL Database. You can manually add client IPs to open the firewall to allow other users to access the database also.

Although you use the Azure portal to view the information available from Query Performance Insight, you must use SQL Server Management Studio to first enable Query Store on a SQL Database. To do this, click Connect in Object Explorer, type the full name

of the server (such as mysampleserver2016.database.windows.net), select SQL Server Authentication in the Authentication drop-down list, type the admin login and password that you created for the database, and then click Connect.

Of course, before you can analyze queries, you must first execute several queries. Execute the query shown in Listing 4-32 which runs 20 times to create a workload on SQL Database. You must wait at least a couple of hours before you can view the corresponding workload analysis in Azure SQL Database Performance Insight.

LISTING 4-32 Execute SQL Database query multiple times after enabling Query Store

Click here to view code image

```
SELECT
    c.LastName,
    c.FirstName,
    c.CompanyName,
    year(OrderDate) AS OrderYear,
    sum(OrderQty) AS OrderQty,
    p.Name AS ProductName,
    sum(LineTotal) AS SalesTotal
FROM SalesLT.SalesOrderHeader soh
JOIN SalesLT.SalesOrderDetail sod ON
    soh.SalesOrderID = sod.SalesOrderID
JOIN SalesLT.Customer c ON
    soh.CustomerID = c.CustomerID
JOIN SalesLT.Product p ON
    sod.ProductID = p.ProductID
GROUP BY
    c.LastName,
    c.FirstName,
    c.CompanyName,
    year(OrderDate),
    p.Name
ORDER BY
    c.CompanyName,
    c.LastName,
    c.FirstName,
    p.Name;
GO 20
```

When Query Store is unable to collect new data, you might see the following messages:

- "Query Store is not properly configured on this database. Click here to learn more."
- "Query Store is not properly configured on this database. Click here to change settings."

There are two ways to clear these messages. First, you can increase the Query Store size or clear Query Store. Second, you can change the Retention and Capture policy and enable Query Store by executing the ALTER DATABASE commands described in the "Create efficient query plans using Query Store" section earlier in this chapter.

When enough time has passed after queries have executed, open the Azure portal, select SQL databases in the navigation pane, click the database to analyze, and then select Query Performance Insight in the Support + Troubleshooting category. The Query Performance Insight blade includes the following three tabs:

- Resource consuming queries
- Long running queries
- Custom

Resource consuming queries

The Resource Consuming Queries tab lists the queries consuming the most resources for the last 24 hours. The top of this section shows a line chart that by default shows the percentage of CPU over time consumed by various queries, as shown in Figure 4-32, although the values for the individual queries are too low to see clearly on the chart. You can click Data IO or Log IO in the top left section to review the relative resource consumption for these resources instead of CPU.

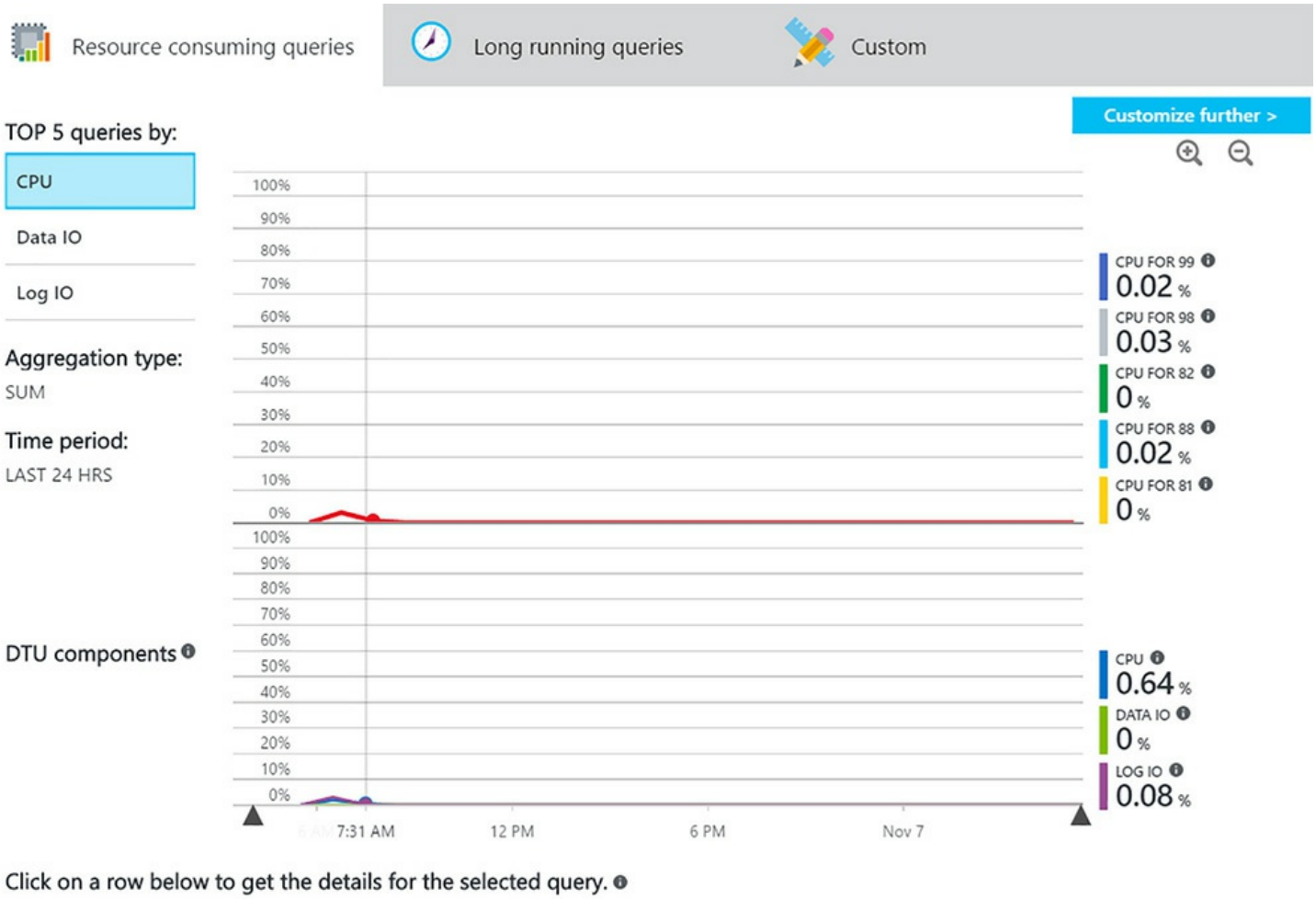


FIGURE 4-32 Resource Consuming Queries chart showing CPU and DTU components consumption for top resource consuming queries

This chart has many interactive features. For example, you can click on the red line on the chart to show the overall *Database Throughput Unit (DTU)* values, as shown in Figure 4-33. A DTU is a single metric to represent CPU, memory, and IO thresholds. You can also change the basis for determining the top five queries from CPU as shown to Data IO or Log IO by clicking the respective filter labels in the top left of the chart. Another option is to zoom in on a period of time by using the sliders (as indicated by the triangles along the horizontal axis of the chart) to increase or decrease the period of time to view in the chart.

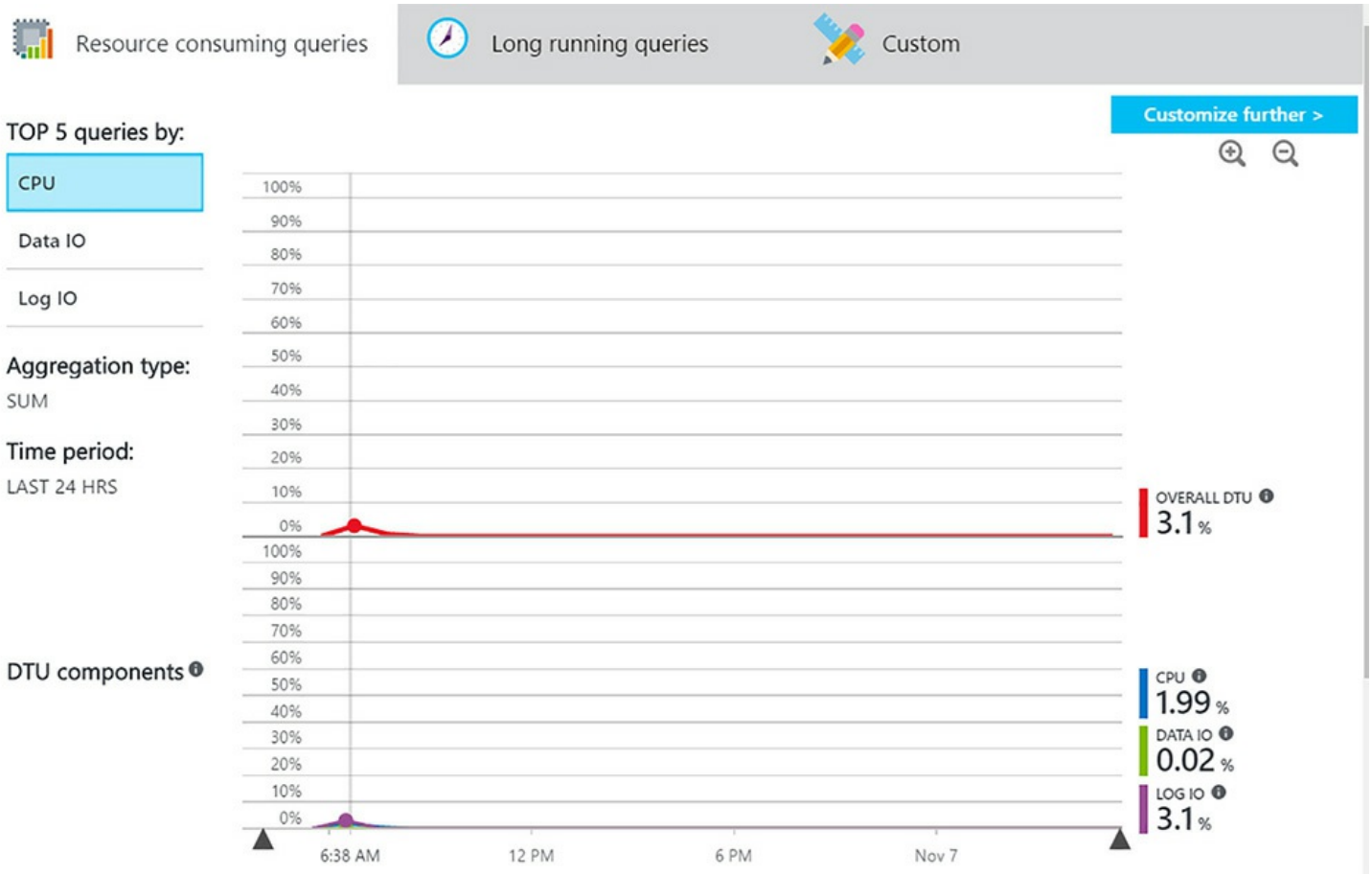


FIGURE 4-33 Resource Consuming Queries chart showing Overall DTU and DTU components consumption

When you scroll down this page, you can view the average CPU, Data IO, Log IO, duration, and execution count for the top 5 queries for a table, as shown in Figure 4-34.

Click on a row below to get the details for the selected query.

QUERY ID	CPU[%]	DATA IO[%]	LOG IO[%]	DURATION[HH:MM:SS]	EXECUTIONS COUNT	#
99	0	0	0	00:00:00.070	6	✓
98	0	0	0	00:00:00.060	6	✓
82	0	0	0	00:00:00.060	46	✓
88	0	0	0	00:00:00.080	29	✓
81	0	0	0	00:00:00.030	279	✓

FIGURE 4-34 Query Performance Insight table listing average metrics for individual queries

When you click a query in this list, you can view its related details in a set of charts, as shown in Figure 4-35.

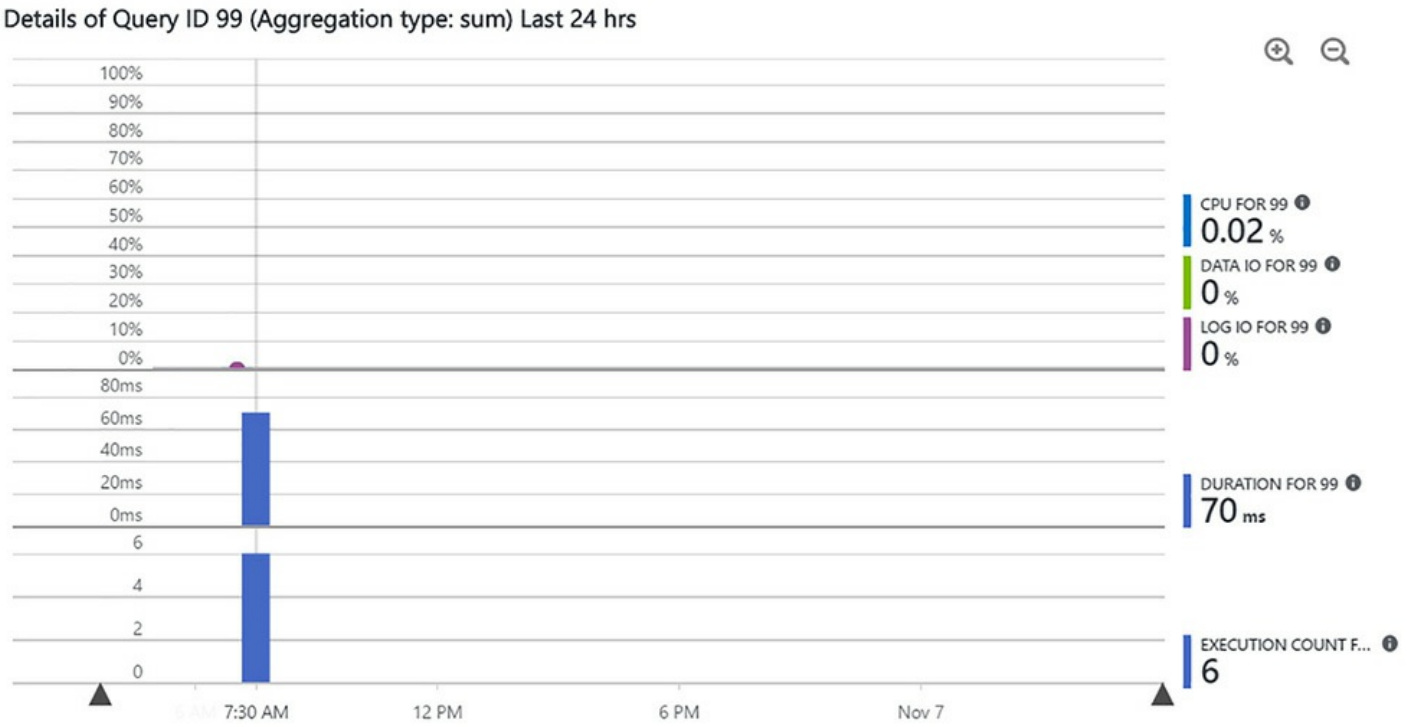


FIGURE 4-35 Query Performance Insight chart showing metrics for a selected query

Below the charts, a table of time intervals displays the metrics, duration, and execution count for the selected query, as shown in Figure 4-36.

INTERVAL	CPU[%]	DATA IO[%]	LOG IO[%]	DURATION[HH:MM:SS]	EXECUTIONS C...
11/6: 05 AM - 06 AM	0	0	0		0
11/6: 06 AM - 07 AM	0	0	0		0
11/6: 07 AM - 08 AM	0.02	0	0	00:00:00.070	6

FIGURE 4-36 Query Performance Insight table showing metrics for a selected query at different time intervals

Long running queries

The Long Running Queries tab, shown in Figure 4-37, shows the top queries based on duration that have executed on the SQL Database during the past 24 hours. The top of the page shows a chart of each query's duration by execution time and the bottom of the page shows a table of the key performance metrics for each query. You can use this information to find queries that might require tuning.

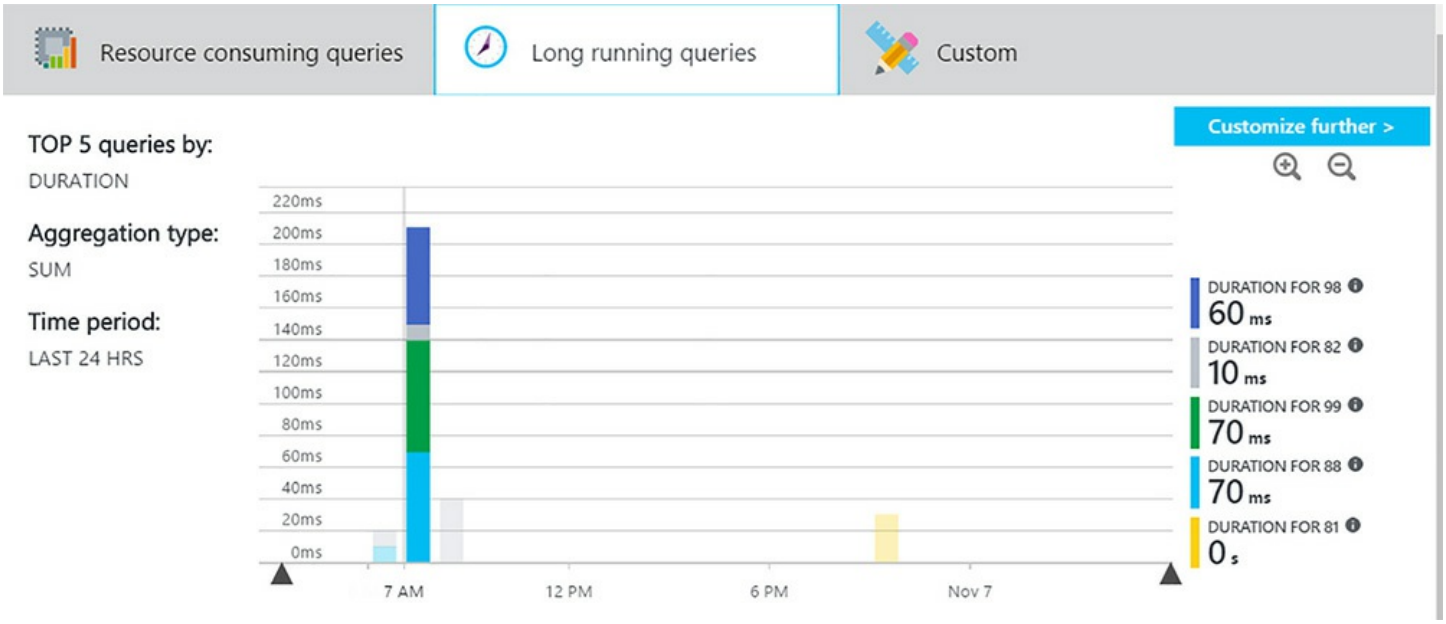


FIGURE 4-37 Query Performance Insight chart showing durations of the top 5 long running queries

Custom

You can also optionally configure a custom view by selecting the Custom tab and then selecting values in each of the following drop-down lists:

- **Metric type** Select one of the following metrics by which to determine top queries: CPU, Data IO, Log IO, Duration, or Execution Count.
- **Time interval** Select one of the following time intervals to set as boundaries for selecting top queries: last 6 hours, 24 hours, past week, past month, and a custom range.
- **Number of queries** Select one of the following numbers to use when selecting top queries: 5, 10, or 20.
- **Aggregate function** Select one of the following aggregate functions to use when aggregating metric values: Sum, Max, or Avg.

Skill 4.3: Manage performance for database instances

SQL Server 2016 and Azure SQL Database include many features that help you monitor and manage the performance of database instances. In this section, we review your options for allocating and optimizing server resources. In addition, we introduce DMVs and SQL Server performance counters that you can use to monitor and troubleshoot database performance over time.

This section covers how to:

- Manage database workload in SQL Server
- Design and implement Elastic Scale for Azure SQL Database
- Select an appropriate service tier or edition
- Optimize database file and tempdb configuration
- Optimize memory configuration
- Monitor and diagnose scheduling and wait statistics using dynamic management objects
- Troubleshoot and analyze storage, IO, and cache issues
- Monitor Azure SQL Database query plans

Manage database workload in SQL Server

The SQL Server Resource Governor helps you manage database workloads by setting limits for the amount of CPU, IO, and memory that incoming requests can consume. Within Resource Governor, a workload is a set of queries or requests for which SQL Server should consistently allocate a specific set of resources. This capability is useful when you are managing multiple tenants on the same server and need to minimize the impact of one tenant's workload on the other tenants' workloads or when you need to track resource consumption by workload for chargeback purposes, just to name two examples.

Important Editions supporting Resource Governor

Resource Governor is supported only in the Enterprise, Developer, or Evaluation editions of SQL Server.

Figure 4-38 shows the relationship between several components managed by Resource Governor. A *resource pool* defines the physical resources of the server and behaves much like a virtual server. SQL Server creates an internal pool and a default pool during installation, and you can add user-defined resource pools. You associate one or more *workload groups*, a set of requests having common characteristics, to a resource pool. As SQL Server receives a request from a session, the *classification* process assigns it to the

workload group having matching characteristics. You can fine-tune the results of this process by creating classifier user-defined functions.

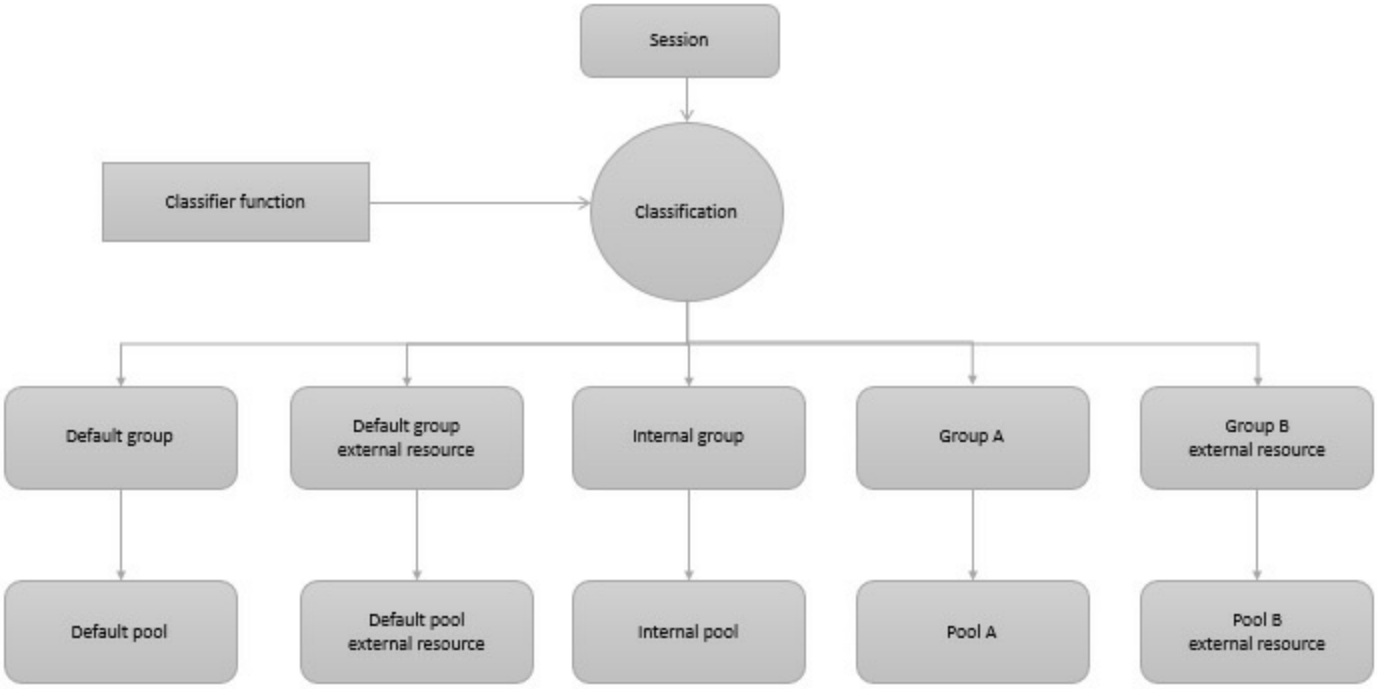


FIGURE 4-38 Resource Governor components

You must enable Resource Governor to start using it. You can do this in SQL Server Management Studio by expanding the Management node in Object Explorer, right-clicking Resource Governor, and selecting Enable. As an alternative, you can execute the following T-SQL statement:

Click here to view code image

```
ALTER RESOURCE GOVERNOR RECONFIGURE;
GO
```

Exam Tip

For the exam, you should be able to review a Resource Governor configuration and identify which workload group will execute a specified T-SQL statement. To correctly answer this type of question, you should have a thorough understanding of Resource Governor's architecture and configuration.

Resource pools

You distribute the amount of memory, CPU, and IO available to SQL Server among resource pools as a means of reducing contention between workloads. Each resource pool is configured with the following settings (except the external resource pool as described

later in this section): Minimum CPU%, Maximum CPU%, Minimum Memory %, and Maximum Memory %. The sum of Minimum CPU% and of Minimum Memory % for all resources pools cannot be more than 100. These values represent the guaranteed average amount of that resource that each resource pool can use to respond to requests. The Maximum CPU% and Maximum Memory % reflect the maximum average amount for the respective resources. SQL Server can use more than the maximum percentage defined for a resource if it is available. To prevent this behavior, you can configure a hard cap on the resource available to the resource pool.

After you enable Resource Governor, SQL Server has the following types of resource pools:

- **Internal** SQL Server uses the internal resource pool for resources required to run the database engine. You cannot change the resource configuration for the internal resource pool. SQL Server creates one when you enable the Resource Governor.
- **Default** In SQL Server 2016, there is one resource pool for standard database operations and a separate resource pool for external processes such as R script execution. These two resource pools are created when you enable the Resource Governor.
- **External** An external resource pool is a new type for SQL Server 2016 that was added to support R Services. Because the execution of R scripts can be resource-intensive, the ability to manage resource consumption by using the Resource Governor is necessary to protect normal database operations. In addition, you can add an external resource pool to allocate resources for other external processes. The configuration for an external resource pool differs from the other resource pool types and includes only the following settings: Maximum CPU%, Maximum Memory %, and Maximum Processes.
- **User-defined resource pool** You can add a resource pool to allocate resources for database operations related to a specific workload.

> **Note Maximum number of supported resource pools per instance**
> SQL Server supports a maximum of 64 resource pools per instance.

You can use the Resource Governor node in Object Explorer to open a dialog box and add or configure resource pools as needed, although this interface does not include all settings available to configure by using T-SQL. When you create a resource pool by using T-SQL, as shown in Listing 4-33, you specify any or all arguments for CPU, the scheduler, memory, and I/O operations per second (IOPS).

Listing 4-33 Create user-defined resource pools

Click here to view code image

```
CREATE RESOURCE POOL poolExamBookDaytime
WITH (
    MIN_CPU_PERCENT = 50,
    MAX_CPU_PERCENT = 80,
    CAP_CPU_PERCENT = 90,
    AFFINITY SCHEDULER = (0 TO 3),
    MIN_MEMORY_PERCENT = 50,
    MAX_MEMORY_PERCENT = 100,
    MIN_IOPS_PER_VOLUME = 20,
    MAX_IOPS_PER_VOLUME = 100
);
GO
CREATE RESOURCE POOL poolExamBookNighttime
WITH (
    MIN_CPU_PERCENT =       0,
    MAX_CPU_PERCENT = 50,
    CAP_CPU_PERCENT = 50,
    AFFINITY SCHEDULER = (0 TO 3),
    MIN_MEMORY_PERCENT = 5,
    MAX_MEMORY_PERCENT = 15,
    MIN_IOPS_PER_VOLUME = 45,
    MAX_IOPS_PER_VOLUME = 100
);
GO
ALTER RESOURCE GOVERNOR RECONFIGURE;
GO
```

Need More Review? Additional information regarding resource poolcreation

For more information about using T-SQL to create a resource pool, see "CREATE RESOURCE POOL (Transact-SQL)" at *https://msdn.microsoft.com/en-us/library/bb895329.aspx*.

Workload groups

Resource Governor monitors the resources consumed in aggregate by the sessions in a workload group to ensure consumption does not exceed the thresholds defined for both the workload group and the resource pool to which it is assigned. The predefined resource pools each have a predefined workload group, but you can also add workload groups to the default, external, and user-defined resource pools.

When you configure a workload group, as shown in Listing 4-34, you can specify the relative importance of a workload group as compared to other workload groups in the same resource pool only. You can also specify the maximum amount of memory or CPU

time that a request in the workload group can acquire from the resource pool, the maximum degree of parallelism (DOP) for parallel requests, or the maximum number of concurrent requests.

Listing 4-34 Create workload groups

Click here to view code image

```
CREATE WORKLOAD GROUP apps
WITH (
     IMPORTANCE = HIGH,
     REQUEST_MAX_MEMORY_GRANT_PERCENT = 35,
     REQUEST_MAX_CPU_TIME_SEC = 0, --0 = unlimited
     REQUEST_MEMORY_GRANT_TIMEOUT_SEC = 60, --seconds
     MAX_DOP = 0, -- uses global setting
     GROUP_MAX_REQUESTS = 1000 --0 = unlimited
)
USING "poolExamBookNighttime";
GO
CREATE WORKLOAD GROUP reports
WITH (
     IMPORTANCE = LOW,
     REQUEST_MAX_MEMORY_GRANT_PERCENT = 25,
     REQUEST_MAX_CPU_TIME_SEC = 0, --0 = unlimited
     REQUEST_MEMORY_GRANT_TIMEOUT_SEC = 60, --seconds
     MAX_DOP = 0, -- uses global setting
     GROUP_MAX_REQUESTS = 100 --0 = unlimited
)
USING "poolExamBookNighttime";
GO
ALTER RESOURCE GOVERNOR RECONFIGURE;
GO
```

Classifier user-defined functions

Resource Governor assigns a request to the default group if there is no criteria that matches the request to a workload group. You must create a user-defined function to provide the criteria necessary to assign a request to a specific workload group. If the user-defined function assigns the request to a non-existent workload group, or if the classification process fails for any reason, Resource Governor assigns the request to the default group.

Let's say that you want to establish a classification function to assign a request to a workload group based on the time of day. Furthermore, you want to use a lookup table for the start and end times applicable to a workload group. Let's start by creating and adding a row to the lookup table, as shown in Listing 4-35. Note that you must create this table in the

master database because Resource Governor uses schema bindings for classifier functions.

Listing 4-35 Create lookup table

Click here to view code image

```
USE master
GO
CREATE TABLE tblClassificationTime  (
    TimeOfDay SYSNAME NOT NULL,
    TimeStart TIME NOT NULL,
    TimeEnd   TIME NOT NULL
) ;
GO
INSERT INTO tblClassificationTime
VALUES('apps', '8:00 AM', '6:00 PM');
GO
INSERT INTO tblClassificationTime
VALUES('reports', '6:00 PM', '8:00 AM');
GO
```

Next, you create the classifier function that uses the lookup table to instruct the Resource Governor which workload group to use when classifying an incoming request. An example of such a classifier function is shown in Listing 4-36.

Listing 4-36 Create and register classifier function

Click here to view code image

```
USE master;
GO
CREATE FUNCTION fnTimeOfDayClassifier()
RETURNS sysname
WITH SCHEMABINDING  AS
BEGIN
    DECLARE @TimeOfDay sysname
    DECLARE @loginTime time
    SET @loginTime = CONVERT(time,GETDATE())
    SELECT
        TOP 1 @TimeOfDay = TimeOfDay
    FROM dbo.tblClassificationTime
    WHERE TimeStart <= @loginTime and TimeEnd >= @loginTime
    IF(@TimeOfDay IS NOT NULL)
        BEGIN
            RETURN @TimeOfDay
```

```
        END
    RETURN N'default'
END;
GO
ALTER RESOURCE GOVERNOR with (CLASSIFIER_FUNCTION =
dbo.fnTimeOfDayClassifier);
ALTER RESOURCE GOVERNOR RECONFIGURE;
GO
```

Note Resources for checking configuration of Resource Governor components

You can confirm the configuration of resource pools, workload groups, and user-defined classifier functions by querying the sys.resource_governor_resource_pools, sys.resource_governor_workload_groups, and sys.resource_governor_configuration system tables respectively. For more information, see "Create and Test a Classifier User-Defined Function" at *https://msdn.microsoft.com/en-us/library/cc645892.aspx*.

Resource Governor management queries

After you configure all the components necessary for Resource Governor, you can monitor resource consumption by using any of the queries shown in Listing 4-37.

Listing 4-37 Monitor resource consumption

Click here to view code image

```
--Current runtime data
SELECT * FROM sys.dm_resource_governor_resource_pools;
GO

SELECT * FROM sys.dm_resource_governor_workload_groups;
GO

--Determine the workload group for each session
SELECT
    s.group_id,
    CAST(g.name as nvarchar(20)) AS WkGrp,
    s.session_id,
    s.login_time,
    CAST(s.host_name as nvarchar(20)) AS Host,
    CAST(s.program_name AS nvarchar(20))  AS Program
FROM sys.dm_exec_sessions s
```

```
INNER JOIN sys.dm_resource_governor_workload_groups g
    ON g.group_id = s.group_id
ORDER BY g.name ;
GO

SELECT
    r.group_id,
    g.name,
    r.status,
    r.session_id,
    r.request_id,
    r.start_time,
    r.command,
    r.sql_handle,
    t.text
FROM sys.dm_exec_requests r
INNER JOIN sys.dm_resource_governor_workload_groups g
    ON g.group_id = r.group_id
CROSS APPLY sys.dm_exec_sql_text(r.sql_handle) AS t
    ORDER BY g.name
GO

-- Determine the classifier running the request
SELECT
    s.group_id,
    g.name,
    s.session_id,
    s.login_time,
    s.host_name,
    s.program_name
FROM sys.dm_exec_sessions s
INNER JOIN sys.dm_resource_governor_workload_groups g
    ON g.group_id = s.group_id  AND
        s.status = 'preconnect'
ORDER BY g.name;
GO

SELECT
    r.group_id,
    g.name,
    r.status,
    r.session_id,
    r.request_id,
    r.start_time,
    r.command,
    r.sql_handle,
    t.text
```

```
FROM sys.dm_exec_requests r
INNER JOIN sys.dm_resource_governor_workload_groups g
    ON g.group_id = r.group_id
        AND r.status = 'preconnect'
CROSS APPLY sys.dm_exec_sql_text(r.sql_handle) AS t
ORDER BY g.name;
GO
```

Design and implement Elastic Scale for Azure SQL Database

Elastic Scale is a feature in SQL Database that you use to adjust the database capacity to match the scalability requirements for different applications. In other words, you can grow or shrink the database by using a technique known as *sharding*, which partitions your data across identically structured database. Sharding is useful when the application data in aggregate exceeds the maximum size supported by SQL Database or when you need to separate data by geography for compliance, latency, or geopolitical reasons.

Although sharding is not a new concept, it requires the use of custom code to create and manage sharded applications and adds complexity to your solution architecture. Elastic Scale provides an elastic database client library and a Split-Merge service that help simplify the management of your applications. That way you can adapt the capacity of SQL Database to support varying workloads and ensure consistent performance without manual intervention.

Elastic database client library

You must use the elastic database client library to implement standard sharding patterns in a SQL Database by calling its features in your elastic scale application. You use it to perform operations across the all shards as a unit or to perform operations on individual shards, as shown in Figure 4-39. The elastic database client library provides the following features:

- **Shard map management** You first register each database as a shard, and then define a shard map manager that directs connection requests to the correct shard by using a *sharding key* or a key range. A sharding key is data such as a customer ID number that the database engine uses to keep related transactions in one database.
- **Data-dependent routing** Rather than define a connection in your application, you can use this feature to automatically assign a connection to the correct shard.
- **Multishard querying** The database engine uses this feature to process queries in parallel across separate shards and then combine the results into a single result set.
- **Shard elasticity** This feature monitors resource consumption for the current workload and dynamically allocates more resource as necessary and shrinks the database to its normal state when those resources are no longer required.

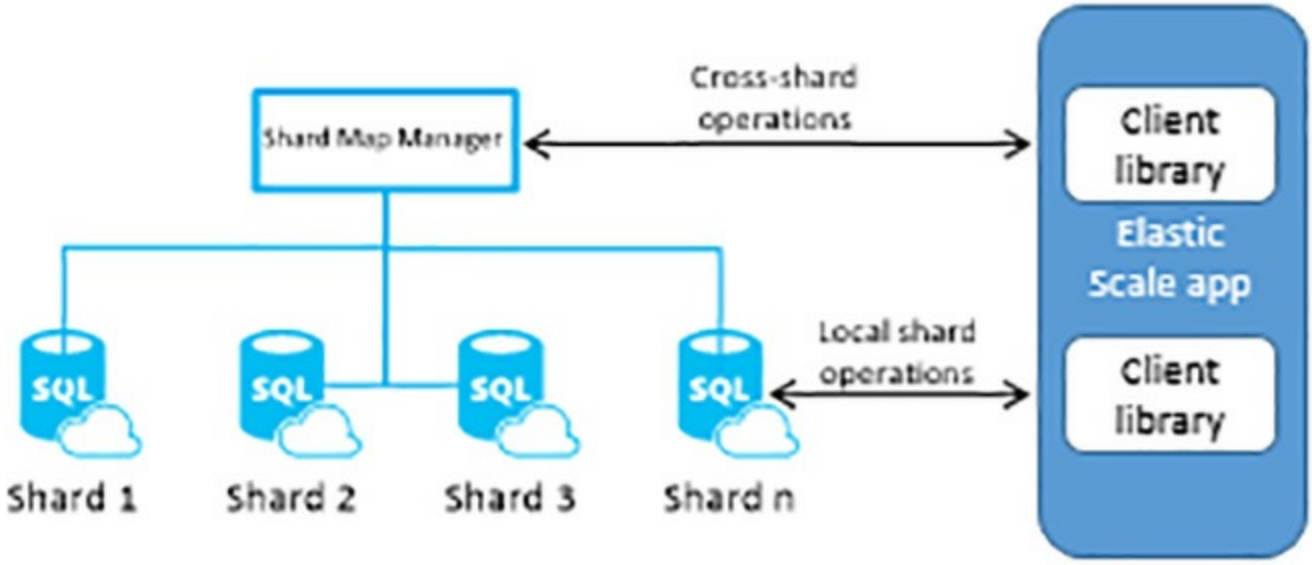


FIGURE 4-39 Sharding management with elastic database client library

> **Note Learning more about working with the elastic database client library**
>
> To use the elastic database client library, you must use Visual Studio 2012 (or higher), C#, and Nuget 2.7 (or higher). You can learn more about working with this client library at "Get started with Elastic Database tools" at *https://azure.microsoft.com/en-us/documentation/articles/sql-database-elastic-scale-get-started/*.

Split-Merge service

You use the Split-Merge service to add or remove databases as shards from the shard set and redistribute the data across more or fewer shards, as shown in Figure 4-40. As demand increases, you can split the data out across a greater number of shards. Conversely, you can merge the data into fewer shards as demand lowers.

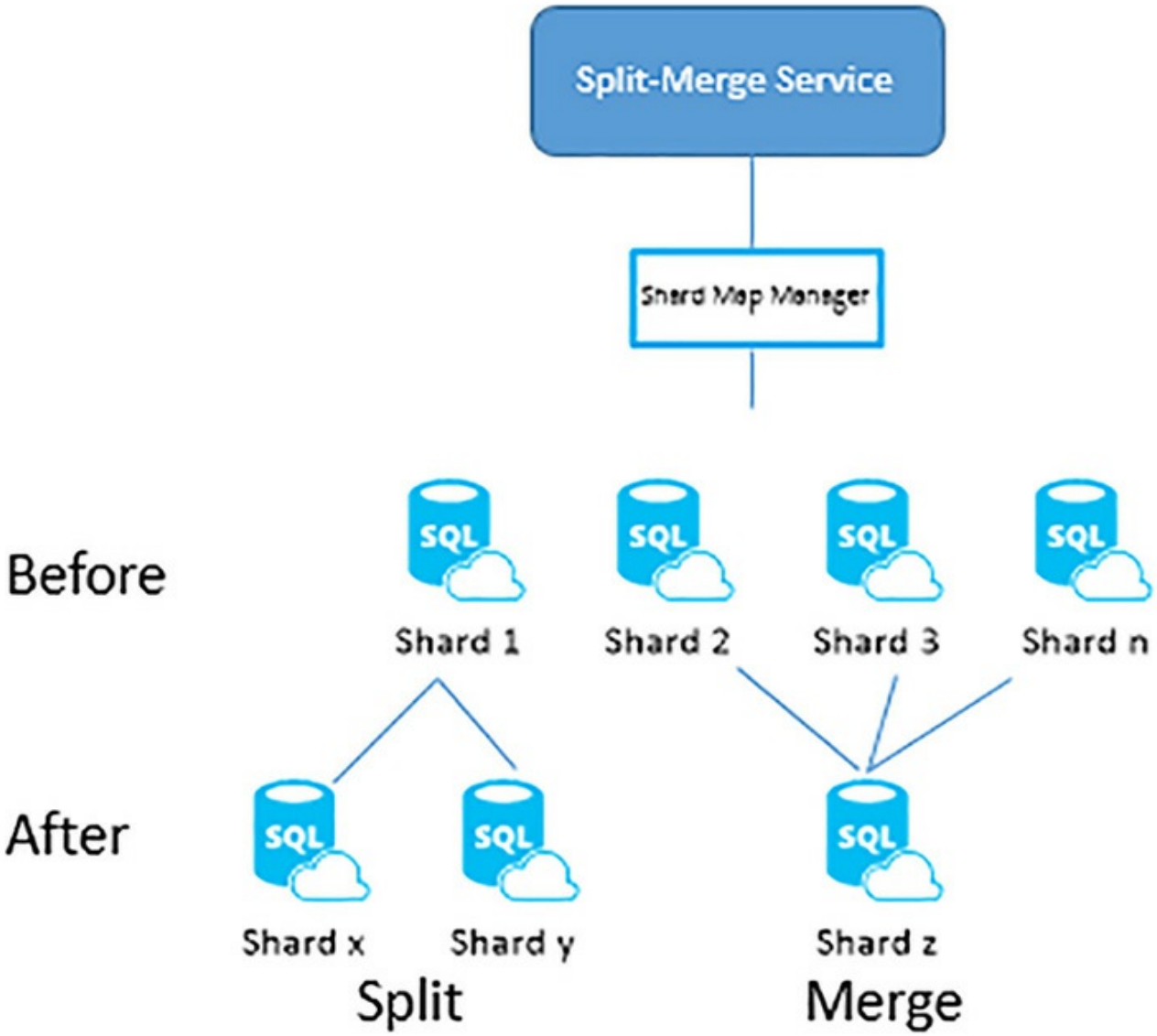


FIGURE 4-40 Examples of splitting and merging shards

Note A tutorial for using the split-merge service

A tutorial that includes a link for the split-merge tool and instructions for using it is available at "Deploy a split-merge service," *https://docs.microsoft.com/en-us/azure/sql-database/sql-database-elastic-scale-configure-deploy-split-and-merge*.

Need More Review? Elastic Scale resources online

For the exam, you should be familiar with the architecture, features, and tools

of Elastic Scale and understand use cases. For more details about Elastic Scale, refer to "Scaling out with Azure SQL Database" at *https://docs.microsoft.com/en-us/azure/sql-database/sql-database-elastic-scale-introduction*.

Select an appropriate service tier or edition

Microsoft uses editions for SQL Server 2016 and service tiers for Azure SQL Database to provide combinations of product features, performance, and price levels so that you can select one that best meets your application's requirements. You should be familiar with the general features and limitations of each edition and understand the differences between each edition for the exam.

SQL Server 2016 is available in the following editions:

- **Express** This edition is a free version of SQL Server with limited features that you can use for small applications and Web sites. The maximum database size supported by this edition is 10 GB. It uses up to 1 GB memory and to the lesser of 1 physical processor or 4 cores. There are three types of SQL Server 2016 Express from which to choose:
  - **LocalDB** You use *LocalDB* for a simple application with a local embedded database that runs in single-user mode.
  - **Express** You use Express when your application requires a small database only and does not require any other components packaged with SQL Server in the Standard edition. You can install this edition on a server and then enable remote connections to support multiple users.
  - **Express with Advanced Services** This edition includes the database engine as well as Full Text Search and Reporting Services.
- **Web** This edition is scalable up to 64 GB of memory and the lesser of 4 physical processors or 16 cores with a maximum database size of 524 PB. It includes the database engine, but without support for availability groups and other high-availability features. It also does not include many of the advanced security and replication features available in Standard or Enterprise edition, nor does it include the business intelligence components such as Analysis Services and Reporting Services, among others. Web edition is intended for use only by Web hosters and third-party software service providers.
- **Standard** This edition scales to 128 GB of memory and the lesser of 4 physical processors or 24 cores. The maximum database size with Standard edition is 524 PB. This edition includes core database and business intelligence functionality and includes basic high-availability and disaster recovery features, new security features such as row-level security and dynamic data masking, and access to non-relational data sources by using JSON and PolyBase.

- **Enterprise** This edition includes all features available in the SQL Server platform and provides the highest scalability, greatest number of security features, and the most advanced business intelligence and analytics features. Like Standard edition, Enterprise edition supports a database size up to 524 PB, but its only limits on memory and processor sizes are the maximums set by your operating system.

  To support higher availability, this edition supports up to 8 secondary replicas, with up to two synchronous secondary replicas, in an availability group, online page and file restore, online indexing, fast recovery, mirrored backups, and the ability to hot add memory and CPU.

  For greater performance, Enterprise edition supports in-memory OLTP, table and index partitioning, data compression, Resource Governor, parallelism for partitioned tables, multiple file stream containers, and delayed durability, among other features.

  Enterprise edition includes many security features not found in Standard edition. In particular, Always Encrypted protects data at rest and in motion. Additional security features exclusive to Enterprise edition include more finely-grained auditing, transparent data encryption, and extensible key management.

  Features supporting data warehouse operations found only in Enterprise edition include change data capture, star join query optimizations, and parallel query processing on partitioned indexes and tables.
- **Developer** This edition is for developers that create, test, and demonstrate applications using any of the data platform components available in Enterprise edition. However, the Developer edition cannot be used in a production environment.
- **Evaluation** This edition is a free trial version of SQL Server 2016. You can use this for up to 180 days that you can use to explore all of the features available in Enterprise Edition before making a purchasing decision.

Need More Review? Features and limitations of SQL Server editions

A complete list of the features and limitations of each edition is available at *https://technet.microsoft.com/en-us/windows/cc645993(v=sql.90)*.

When you select the type of SQL Database to implement, you choose a service tier along with a performance level. The service tier sets the maximum database size while the performance level determines the amount of CPU, memory, and IO thresholds which collectively are measured as a DTU. When you create a new SQL Database, you can choose from the following service tiers:

- **Basic** This service tier has a maximum database size of 2 GB and performance level of 5 DTUs. You use this option when you need a small database for an application or website with relatively few concurrent requests. The benchmark transaction rate is 16,600 transactions per hour.

- **Standard** This service tier has a maximum database size of 250 GB and performance levels ranging from 10 to 100 DTUs. You can use this option when a database needs to support multiple applications and multiple concurrent requests for workgroup and web applications. With 50 DTUs, the benchmark transaction rate is 2,570 transactions per minute.
- **Premium** This service tier has a maximum database size of 1 TB and performance levels ranging from 125 to 4,000 DTUs. You use this option for enterprise-level database requirements. With 1,000 DTUs, the benchmark transaction rate is 735 transactions per second.

Optimize database file and tempdb configuration

One option that you have for improving the performance of read and write operations is to optimize the configuration of files that SQL Server uses to store data and log files. Your optimization goal is to reduce contention for storage and IO of files used not only by your database, but also by tempdb.

Database file optimization

When your application must support a high volume of read/write operations, you should consider taking the following steps to mitigate disk contention:

- **File placement** Data and log files should be placed on separate physical disks for better performance. Remember from the explanation of transactions that we described in Chapter 3, "Managing database concurrency," that SQL Server writes each transaction to the log before updating the data file. By separating the two file types, the read/write head for the disk with the log file can work more efficiently without frequent interruptions by the writes to the data file. Furthermore, consider using a disk with high write performance for the log file. This recommendation is less applicable when the bulk of SQL Server activity is read operations.

  By default, the data and log files for a new database are placed on the same drive, and normally in the same directory as the system databases: \Program Files\Microsoft SQL Server\MSSQL13.MSSQLSERVER\MSSQL\DATA. You can move the data and log files to a new location by using the ALTER DATABASE command as shown in Listing 4-38, replacing <databasename> and <drive:\filepath> with the appropriate names for your database and folder structures.

LISTING 4-38 Relocate data and log files

Click here to view code image

```
ALTER DATABASE <databasename>
SET OFFLINE;
GO
```

```
ALTER DATABASE <databasename>
MODIFY FILE (NAME = <databasename>_Data, FILENAME = "
<drive:\filepath>\
Data\<databasename>_Data.mdf");
GO
ALTER DATABASE <databasename>
MODIFY FILE (NAME = <databasename>_Log, FILENAME =
"drive:filepath>\
Log\<databasename>_Log.mdf");
ALTER DATABASE <databasename>
SET ONLINE;
```

Another benefit of separating data and log files on separate drives is mitigating a potential failure. If the drive containing the data files fails, you can still access the log file from the other disk and recover data up to the point of failure.

- **File groups and secondary data files** By default, each database has a primary filegroup that contains the primary data file containing system tables and database files created without a filegroup specification. SQL Server uses the primary filegroup as the default for new indexes and tables that you create without placing them in a specific filegroup, but you can create a new filegroup and designate it as the default. There can only be one default filegroup, but you can create as many additional file groups as you need as containers for one or more data and log files. Files within a single filegroup can be spread across multiple disks to increase the parallelism of data access. Furthermore, you can separate tables or indexes that are heavily accessed from lesser used tables or indexes by assigning them to different filegroups, which in turn are each placed on separate disks. Listing 4-39 shows how to create a database with multiple filegroups and how to add a filegroup to an existing database to isolate an index on its own disk:

LISTING 4-39 Create and alter database with multiple file groups

Click here to view code image

```
--Create a database on 4 drives
CREATE DATABASE DB1 ON
PRIMARY
    (Name = <databasename>, FILENAME = '<drive1:filepath>\
<databasename>.mdf'),
    FILEGROUP FGHeavyAccess1
    (Name = <databasename>_1, FILENAME = '<drive3:filepath>\
<databasename>_1.ndf')
LOG ON
    (Name = <databasename>_1_Log, FILENAME =
'<drive3:filepath>\<databasename>_1_
```

```
log.ldf'),
    (Name = <databasename>_1, FILENAME = '<drive4:filepath>\
<databasename>_1_
log_2.ldf');
-- Add filegroup for index
ALTER DATABASE <databasename>
    ADD FILEGROUP FGIndex;
--  Add data file to the new filegroup
ALTER DATABASE <databasename>
ADD FILE (
    NAME = <databasename>,
    FILENAME = '<drive1:filepath>\<databasename>.ndf',
    SIZE=1024MB,
    MAXSIZE=10GB,
    FILEGROWTH=10%)
TO FILEGROUP FGIndex;
-- Add index to filegroup
CREATE NONCLUSTERED INDEX ix_Example
    ON Examples.BusyTable(TableColumn)
   ON FGIndex;
```

- **Partitioning** You can use partitioning to place a table across multiple filegroups. Each partition should be in its own filegroup to improve performance. To map a value in a partition function to a specific filegroup, you use a partition scheme. Let's say you have four filegroups—FGYear1, FGYear2, FGYear3, and FGYear4—and a partition function PFYearRange that defines four partitions for a table. You can create a partition schema to apply the partition function to these filegroups as shown in Listing 4-40.

LISTING 4-40 Create partition scheme to map partition function to filegroups

Click here to view code image

```
CREATE PARTITION SCHEME PSYear
    AS PARTITION PFYearRange
    TO (FGYear1, FGYear2, FGYear3, FGYear4);
```

tempdb optimization

Because so many operations, such as cursors, temp tables, and sorts, to name a few, rely on tempdb, configuring tempdb properly is critical to the performance of the database engine. Consider performing the following steps to optimize tempdb configuration:

- **SIMPLE recovery model** By using the SIMPLE recovery model, which is the default, SQL Server reclaims log space automatically so that the space required for

the database is kept as low as possible.

- **Autogrowth** You should keep the default setting which allows tempdb files to automatically grow as needed.
- **File placement** The tempdb data and log files should be placed on different disks than your production database data and log files. Do not place the tempdb data files on the C drive to prevent the server from failing to start after running out of hard drive space. In addition, be sure to place the tempdb log file on its own disk. Regardless, put tempdb files on fast drives.
- **Files per core** In general, the number of data files for tempdb should be a 1:1 ratio of data files to CPU cores. In fact, in SQL Server 2016, the setup wizard now assigns the correct number based on the number of logical processors that it detects on your server, up to a maximum of 8, as shown in Figure 4-41.
- **File size** When you configure the database engine at setup, the default file size recommended by the setup wizard of 8 MB for an initial size with an autogrowth setting of 64MB is conservative and too small for most implementations. Instead, consider starting with an initial size of 4,096 MB with an autogrowth setting of 512 MB to reduce contention and minimize the impact of uncontrolled tempdb growth on performance. If you dedicate a drive to tempdb, you can set up the log files evenly on the drive to avoid performance issues caused by SQL Server pausing user activity as it grows the log files.

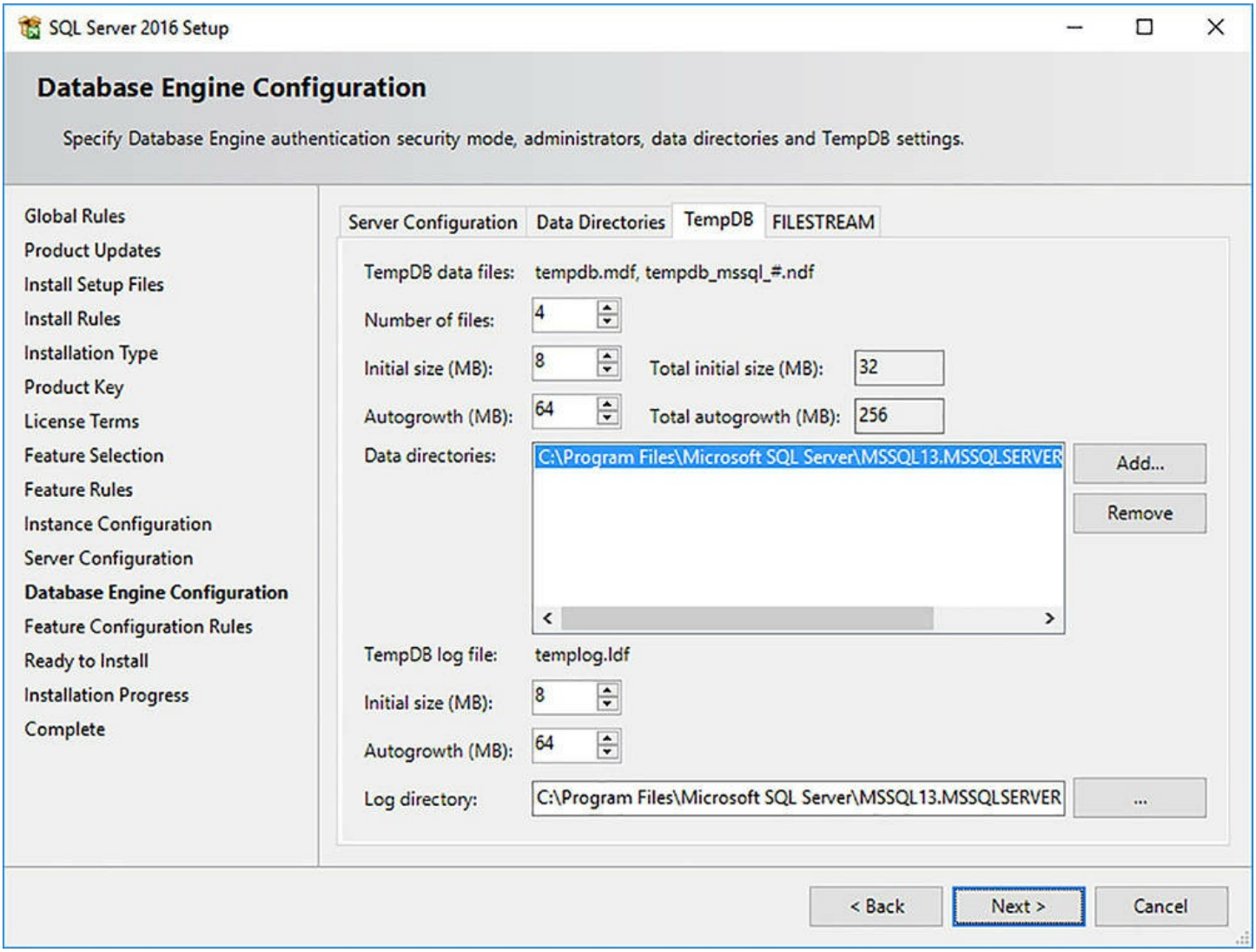


FIGURE 4-41 TempDB configuration in SQL Server 2016 Setup wizard

> **Need More Review? Optimization and capacity planning for tempdb**
>
> For specific size and placement recommendations, see "Optimizing tempdb Performance" at *https://technet.microsoft.com/en-us/library/ms175527.aspx* and "Capacity Planning for tempdb" at *https://technet.microsoft.com/en-us/library/ms345368.aspx*.

Optimize memory configuration

SQL Server's memory manager dynamically allocates memory according to the workloads on the host computer and in the database engine. However, you can use the following server configuration options to optimize SQL Server memory:

- **min server memory** Use this option to prevent SQL Server from releasing memory to the operating system when the server memory drops to this threshold.
- **max server memory** Use this option to ensure that other applications running on the same computer as SQL Server have adequate memory. When an application requests

memory only as needed, you do not need to configure this option. It applies only when an application uses the memory available when it starts and does not later request more memory when necessary. You should configure this option to prevent SQL Server from taking the memory that the application might need.

- **max worker threads** Use this configuration to define the number of threads available to user operations. If you keep the default value of 0, SQL Server configures the number of worker threads each time the service restarts.
- **index create memory** Use this option to set the maximum amount of memory that SQL Server initially allocates for index creation. SQL Server will allocate more memory later if necessary, but only if it is available. Typically, you do not need to configure this option, but if SQL Server is experiencing performance delays related to indexing, you can increase the value of this option.
- **min memory per query** Use this option to improve performance of memory-intensive queries by establishing the minimum amount of memory allocated for query execution. SQL Server can use more memory than the configured minimum if it is available.

To configure memory, right-click the server instance in Object Explorer and select Properties. Click the Memory page, and then in the Server Memory Options, type the appropriate values for any property except *max worker threads*. You can also use T-SQL to adjust a property value like this:

Click here to view code image

```
EXEC sp_configure 'show advanced options', 1;
GO
RECONFIGURE;
GO
EXEC sp_configure 'min memory per query', 512 ;
GO
RECONFIGURE;
GO
```

Monitor and diagnose schedule and wait statistics using dynamic management objects

One of the best ways to determine which SQL Server resource is a bottleneck on performance is to review wait statistics. To better understand why waits, it is helpful first to understand how SQL Server manages incoming requests. Each authenticated connection is assigned to a session by SQL Server which then uses a pseudo-operating system called the SQL Operating System (SQLOS) Scheduler to schedule CPU time for each session's requests. There is one SQLOS Scheduler per logical CPU core on the server to manage the worker threads performing operations necessary to complete a request. These worker threads must work cooperatively by running only for 4-milliseconds, known as a *quantum*,

before yielding the CPU to another worker thread and waiting in a runnable queue for another turn. It might voluntarily yield the CPU if its quantum has not yet expired and it cannot complete its task, because a resource it needs is unavailable. In this case, the worker thread is moved to a waiter list and then later moves back to the runnable queue when the needed resource becomes available.

Wait statistics allow you to analyze the time a worker thread spends in various states before it completes a request by providing the following key pieces of information:

- **Wait type** The cause of the wait. For example, the disk IO is slow, resources are locked, CPU is under pressure, an index is missing, or many other reasons. There are hundreds of wait types that SQL Server tracks.
- **Service time** The amount of time that a thread runs on the CPU.
- **Wait time** The amount of time that a thread is not running because it is in the waiter list.
- **Signal wait time** The amount of time that a thread is in the runnable queue ready to run, but waiting for CPU time.
- **Total wait time** The sum of wait time and signal wait time.

You can access wait statistics through the following DMVS:

- **sys.dm_os_wait_stats** View information about completed waits at the instance level.
- **sys.dm_exec_session_wait_stats** View information about waits at the session level.
- **sys.dm_os_waiting_tasks** View information about requests in the waiter list.

sys.dm_os_wait_stats

In Chapter 3 we explored how to use this DMV for troubleshooting lock issues, but it is also useful for discovering the most frequently occurring waits since the last reset of the cumulative values. The cumulative wait time in this DMV includes the cumulative signal wait time, so subtract signal wait time from wait time when you want to determine the cumulative time threads spend in the waiter list.

> **Note Isolating top waits**
>
> You can find a useful query for isolating top waits to help focus your troubleshooting efforts in Paul Randal's blog post, "Wait statistics, or please tell me where it hurts" at *http://www.sqlskills.com/blogs/paul/wait-statistics-or-please-tell-me-where-it-hurts/*.

By reviewing this DMV for specific characteristics, you can uncover some of the following potential issues on your server:

- **CPU pressure** Compare the signal wait time to the total wait time to determine the

relative percentage of time that a thread has to wait for its turn to run on the CPU. When this value is relatively high, it can be an indicator that the CPU is overwhelmed by queries that require tuning or your server needs more CPU. You can confirm whether the issue is related to CPU by checking the runnable_tasks_count column in the sys.dm_os_schedulers DMV to see if there is a high number of tasks in the runnable queue. You might also see a higher occurrence of the SOS_SCHEDULER_YIELD wait type if the CPU is under pressure. In addition, you can monitor CPU-related performance counters as described in Skill 4.4.

- **IO issues** If tasks are waiting for the IO subsystem, you will see waits that contain *IO* in the name. In particular, monitor the trend in average wait time which is calculated by dividing wait_time_ms by waiting_tasks_count. If it starts trending upward, investigate IO using performance counters.

  Two wait types that will appear frequently in this DMV when IO issues exist are ASYNC_IO_COMPLETION and IO_COMPLETION. Check physical disk performance counters to confirm this diagnosis, which we describe in the next section, "Troubleshoot and analyze storage, IO, and cache issues." Consider adding indexes to reduce IO contention.

  You might also see PAGEIOLATCH waits when a thread is waiting for latches to release after writing a data page in memory to disk or WRITELOG waits when the log management system is waiting to flush to disk. These wait types can indicate either an IO subsystem problem or a memory problem. To narrow down the possibilities, you need to check IO statistics by using sys.dm_io_virtual_file_stats and by reviewing IO-related performance counters.

- **Memory pressure** The PAGEIOLATCH wait might also indicate memory pressure instead of an IO subsystem problem. It appears when SQL Server does not have enough free memory available for the buffer pool. Check the Page Life Expectancy performance counter to see if it is dropping as compared to a baseline value to confirm whether memory is the reason for this wait type. If you see an increase in CXPACKET waits in conjunction with PAGEIOLATCH waits, the culprit could be a query plan using large table or index scans.

  Another indicator of a memory pressure issue is the RESOURCE_SEMAPHORE wait. It occurs when a query requests more memory than is currently available. You can check the sys.dm_exec_query_memory_grants DMV and combine it with sys.dm_exec_sql_text and sys.dm_exec_sql_plan DMVs to find the memory-intensive queries and review their query plans.

sys.dm_exec_session_wait_stats

This DMV is new in SQL Server 2016 and is identical in structure to sys.dm_os_wait_stats, but has an additional column for session ID. However, it is important to note that this new DMV only includes information for sessions that are

currently connected. When a session disconnects, its wait statistics are cleared from the DMV. Nonetheless, it can be helpful when you need to diagnose the workload for a specific session.

sys.dm_os_waiting_tasks

We introduced this DMV in Chapter 3 as a tool for finding blocked sessions, but you can also use it to find the requests currently waiting for a resource and why. As one example, you might filter the DMV by using the wait_duration_ms column to find tasks that have been waiting longer than a threshold that you specify.

> **Note Additional resource for using the sys.dm_os_waiting_tasks DMV**
>
> Paul Randal has published a script that demonstrates how to use sys.dm_os_waiting_tasks with other DMVs to gather information about blocked sessions, the wait type, the resource affected, the query text, and the query plan in his blog post "Updated sys.dm_os_waiting_tasks" at *http://www.sqlskills.com/blogs/paul/updated-sys-dm_os_waiting_tasks-script/*.

> **Need More Review? Learning more about analyzing wait statistics**
>
> You can learn more about analyzing wait statistics in Jonathan Kehayias and Erin Stellato's *SQL Server Performance Tuning Using Wait Statistics: A Beginner's Guide*, a free whitepaper at *https://www.simple-talk.com/free-stuff/sql-server-performance-tuning-using-wait-statistics-a-beginners-guide/*.

Troubleshoot and analyze storage, IO, and cache issues

Troubleshooting and analyzing storage, IO, and cache issues is a huge topic to which many books, blog posts, and workshops are dedicated because there are many different ways to configure the disk and IO subsystem and many different issues that can arise. For the exam, you should understand how to get information about storage, IO, and cache performance from SQL Server and the Microsoft Windows operating system.

Storage and IO

Storage bottlenecks occur when your data or log files are stored on slow disks or when the RAID is not configured appropriately for your workload. Unless your application is using memory-optimized tables, SQL Server is frequently reading data from disk in response to queries or writing new or changed data to disk. Meanwhile, tempdb is disk-based and uses a lot of IO for grouping and sorting operations. Conditions within SQL Server that create IO bottlenecks include frequent index scans, inefficient queries, and outdated statistics.

Although SQL Server is requesting the reads and writes, the operating system controls the system bus, disk controller cards, disks, and other IO devices to physically perform the disk IO. Another factor to consider is the demand that other applications running on the server can place on the IO subsystem.

Your first indication that you might have an IO problem might come from an analysis of waits as we described in the previous section. Your next step is to use the sys.dm_io_virtual_file_stats DMV in combination with sys.master_files to analyze cumulative metrics related to each database including its data and log files. These DMVs help you find the busiest files and provides IO stall information that tells you how long users had to wait for IO operations to finish.

> **Note Example of IO subsystem latency analysis**
>
> You can find an example of analyzing these DMVs in Paul Randal's blog post, "How to examine IO subsystem latencies from within SQL Server" at *http://www.sqlskills.com/blogs/paul/how-to-examine-io-subsystem-latencies-from-within-sql-server/*.

Another way to find these issues is to use the sys.dm_os_performance_counters DMV. The availability of the DMV means you can easily get SQL Server-related performance counter information without first setting up Windows Performance Monitor.

> **Note An online resource for querying SQL Server performance counters**
>
> Interpreting the information found in the sys.dm_os_performance_counters information can sometimes be challenging because counter values can be an average of operations executed within a sample interval or the last observed value for a counter, to name only two types of values. Jason Strate has produced a comprehensive explanation of the types of values and how to derive important calculations from counter values when the available from the DMV in his blog post, "Querying Performance Counters in SQL Server," at *http://www.jasonstrate.com/2012/11/querying-performance-counters-in-sql-server/*.

The following performance counters, accessible by executing the statements shown in Listing 4-41, provide insight into the amount of IO that SQL Server is directly contributing to the server:

- **SQLServer:Buffer Manager: Page lookups/sec** Average requests per second at which SQL Server finds a page in the buffer pool. This value should be lower than *SQLServer:SQL Statistics: Batch Requests/sec* multiplied by 100.
- **SQLServer:Buffer Manager: Page reads/sec** Average rate at which SQL Server

reads from disk. This value should be lower than the hardware specifications for the IO subsystem's read operations.

- **SQLServer:Buffer Manager: Page writes/sec** Average rate at which SQL Server writes to disk. This value should be lower than the hardware specifications for the IO subsystem's write operations.

Listing 4-41 Review SQL Server:Buffer Manager performance counters

Click here to view code image

```
SELECT
    object_name,
    counter_name,
    instance_name,
    cntr_value,
    cntr_type
FROM sys.dm_os_performance_counters
WHERE object_name = 'SQLServer:Buffer Manager' AND
    counter_name IN
        ('Page lookups/sec', 'Page reads/sec', 'Page
writes/sec')
```

If these performance counters are too high, consider one or more of the following solutions:

- Tune database performance by adding new indexes, improving existing indexes, or normalizing tables, or partitioning tables.
- Replace the IO subsystem hardware with faster components.

Cache issues

SQL Server is self-tuning and manages memory dynamically. It will use as much memory as you can give it, and it will not release memory until the operating system sets the low memory resource notification flag. Cache bottlenecks occur when SQL Server does not have enough memory to manage. To diagnose cache issues, start by checking the physical memory of the server and identifying how other applications on the server are using memory. You might need to analyze and tune specific queries as well. For example, if an index is missing on a large table, SQL Server must perform a table scan which reads a significant amount of data into memory.

The following DMVs are useful for understanding memory usage on your server:

- **sys.dm_os_memory_cache_counters** View the current state of the cache.
- **sys.dm_os_sys_memory** View resource usage information for the server, including total physical and available memory and high or low memory state.

- **sys.dm_os_memory_clerks** View usage information by memory clerk processes that manage memory for SQL Server.

> **Note Learning more about using DMVs to tune performance**
>
> You can learn more about using these and related DMVs to investigate memory usage in Louis Davidson and Tim Ford's free book, *Performance Tuning with SQL Server Dynamic Management Views*, which you can download from *https://assets.red-gate.com/community/books/performance-tuning-with-dmvs.pdf*.

You should also use the following performance counters to monitor whether SQL Server has adequate memory:

- **SQLServer:Buffer Manager: Free List Stalls/Sec** Number of requests per second that SQL Server waits for a free page in the buffer cache. If this value is greater than zero on a frequent basis, the server is experiencing memory pressure.
- **SQLServer:Buffer Manager: Lazy Writes/Sec** Number of times per second that SQL Server flushes pages to disk. If this number is rising over time, and Free List Stalls/Sec is also greater than zero, you likely need more memory on the server.
- **SQLServer:Memory Manager: Memory Grants Outstanding** Number of processes that have acquired a memory grant successfully. A low value might signify memory pressure.
- **SQLServer:Memory Manager: Memory Grants Pending** Number of processes that are waiting for a memory grant. If this value is greater than zero, consider turning queries or adding memory to the server.

Monitor Azure SQL Database query plans

To monitor Azure SQL Database query plans, you use many of the same techniques that you use to monitor SQL Server query plans. Specifically, you can choose any of the following methods:

- **T-SQL statements** You can use the Showplan SET options in SQL Server Management Studio to capture query plans, just as you can for SQL Server. You can also use the Display Estimated Execution Plan and Include Actual Execution Plan buttons in the toolbar to generate the respective graphical query plan.
- **Extended Events** You can use Extended Events to capture query plans, much like you can for SQL Server. There are some slight differences, however. Instead of using the ON SERVER clause in the CREATE EVENT SESSION, ALTER EVENT SESSION, and DROP EVENT SESSION commands, you must use ON DATABASE instead. If you want to save a query plan to a file, you must write the file to an Azure Storage container. In addition, there are also several DMVS for Extended Events that

are unique to SQL Database.

Note Understanding differences in Extended Events between SQL Server and SQL Database

For more information about the differences in usage of Extended Events between SQL Server and SQL Database, see "Extended events in SQL Database" at *https://docs.microsoft.com/en-us/azure/sql-database/sql-database-xevent-db-diff-from-svr*.

- **Query Store** Query Store is enabled by default for V12 databases. You can access the Query Store views in SQL Server Management Studio in the same way that you do for SQL Server. Also, you can use the same Query Store DMVs.

Important SQL Database lacks support for SQL Trace

The use of SQL Trace is not supported in SQL Database.

Skill 4.4: Monitor and trace SQL Server baseline performance metrics

One of the most important responsibilities of a DBA is to ensure that SQL Server runs smoothly and performs optimally. To fulfill this responsibility, you should be familiar with the array of tools available to help you uncover and diagnose problems occurring on the server. We have introduced several of these tools in the preceding pages of this chapter, but we did so with specific contexts in mind. Now let's step back and survey the tools again in terms of how you can use them to baseline server performance at the operating system and SQL Server instance levels. In addition, we introduce some new tools that you can use towards this same goal.

This section covers how to:

- Monitor operating system and SQL Server performance metrics
- Compare baseline metrics to observed metrics while troubleshooting performance issues
- Identify differences between performance monitoring and logging tools
- Monitor Azure SQL Database performance
- Determine best practice use cases for Extended Events
- Distinguish between Extended Events targets
- Compare the impact of Extended Events and SQL Trace
- Define differences between Extended Events Packages, Targets, Actions, and Sessions

Monitor operating system and SQL Server performance metrics

At any time, without any elaborate setup requirements, you can access the following tools to check performance metrics for the operating system and SQL Server:

- **Dynamic management objects (DMOs)** DMVs and DMFs provide insight into the historical and current state of SQL Server. You can query DMOs on an ad hoc basis, or you can create export the data into tables for long-term storage and trend analysis.
- **Performance Monitor** Operating system and SQL Server performance counters provide useful information to corroborate metrics obtained from other sources and to help narrow down the range of possible causes when investigating specific problems.
- **SQL Trace** You can set up server-side or SQL Server Profiler tracing for a variety of events as a method of investigating problematic workloads and poorly performing queries.
- **Extended Events** You can create Extended Events sessions as a more lightweight approach to server-side tracing.

Dynamic management objects

In Chapter 3 and earlier in this chapter, we explored a wide variety of DMOs that you can use to troubleshoot locking and blocking issues, collect execution statistics on natively compiled stored procedures, review index usage, access Query Store information, review wait statistics, and troubleshoot storage, IO, and cache issues. Did you happen to notice any patterns in the naming of these DMOs? By understanding the naming conventions, you can get a high-level view of the range of information that DMOs supply. For the exam, you do not need to be familiar with the complete list of DMO categories supported in SQL Server. However, given the name of a specific DMO, you should be able to identify the type of information it provides at a general level by its association with one of the following categories:

- **sys.dm_exec_*** Connections, sessions, requests, and query execution
- **sys.dm_os_*** Information for the operating system on which SQL Server runs
- **sys.dm_tran_*** Details about transactions
- **sys.dm_io_*** IO processes
- **sys.dm_db_*** Database-scoped information

> **Note Additional information about DMOs**
>
> For more in-depth information about dynamic management objects, refer to "Dynamic Management Views and Functions (Transact-SQL)" at *https://msdn.microsoft.com/en-us/library/ms188754.aspx*.

Most DMOs provide information about the current state, such as currently blocked sessions in sys.dm_os_waiting_tasks, or information accumulated since SQL Server last restarted, such as sys.dm_os_wait_stats. SQL Server retains the information accessible through DMOs in memory only and does not persist it to disk. Therefore, the information is reset when SQL Server restarts.

Performance Monitor

Performance Monitor, also known as PerfMon, is a tool provided with the Windows operating system that you can use to monitor operating system, application, and hardware performance in real time. You can even establish thresholds and receive alerts when thresholds are crossed. As an alternative, you can capture performance data in logs that you can review in the graphical interface or save to SQL Server tables for trend analysis.

Typically, you use performance counters to confirm suspicions of a problem that you uncover by using wait statistics rather than as a starting point. Like DMOs,

For real-time analysis, open Performance Monitor, and click the Add button in the toolbar. In the Add Counters dialog box, select the server to monitor in the Select Counters From Computer drop-down list (or use the Browse button to locate a server on your network), and then scroll through the Available Counters list to locate the set of counters to monitor, such as PhysicalDisk. When you monitor PhysicalDisk counters, you can select a specific disk to monitor or all disks. Select a specific counter, such as % Disk Time, and click the Add button. Continue adding counters as needed. When finished, click OK. You can then view real-time metrics for the selected counters. If you are monitoring multiple counters at the same time, right-click a counter and select Properties to change Color, Width, Style values to more easily distinguish between counters, as shown in Figure 4-42. You might also need to reset the scale for a counter.

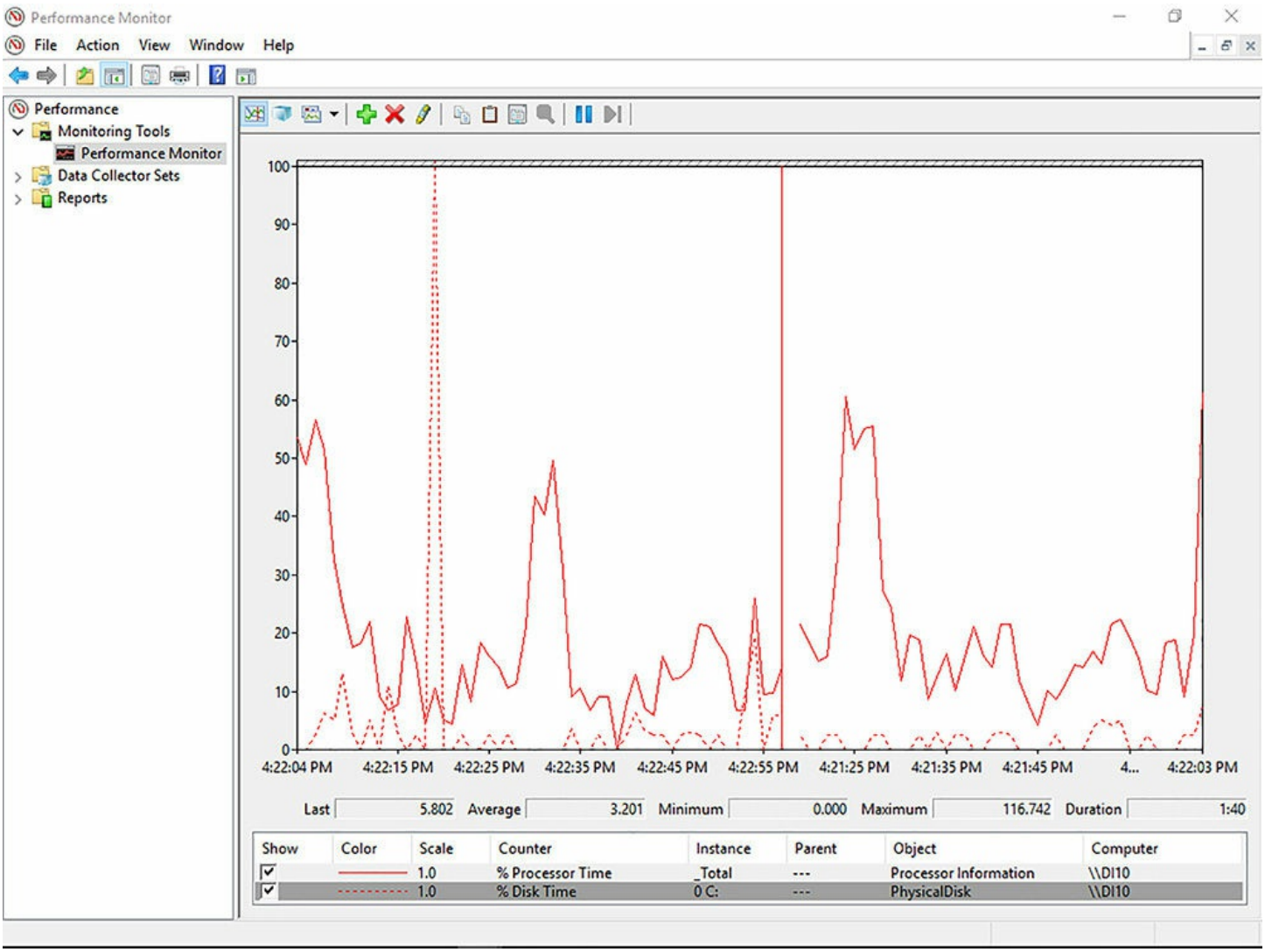


FIGURE 4-42 Real-time monitoring of performance counters in Performance Monitor

To identify issues in disk IO activity, start by reviewing the following performance counters (replacing PhysicalDisk with the LogicalDisk if you have multiple logical partitions on the same disk):

- **PhysicalDisk: % Disk Time** Percentage of time the disk is active with reads and writes. If this percentage is greater than 90 percent, review the *Physical Disk: Current Disk Queue Length* counter.
- **PhysicalDisk: Avg. Disk sec/Read** Average read latency in seconds. This value should be less than 0.20.
- **PhysicalDisk: Avg. Disk sec/Transfer** Average latency of IO requests to the disk in seconds. This value should be less than .020.
- **PhysicalDisk: Avg. Disk sec/Write** Average read latency of IO requests to the disk in seconds. This value should be less than .020.
- **PhysicalDisk: Current Disk Queue Length** Number of IO requests waiting for access to the disk. This value should be no more than two times the number of spindles for the disk. Most disks have a single spindle, but a redundant array of

independent disks (RAID) typically have more than one spindle, even though the RAID device appears as a single physical disk in System Monitor.

- **Memory: Page Faults/sec** This performance counter increases when processes on the server are consuming too much memory and SQL Server must page to disk. Paging itself is not necessarily a cause of an I/O bottleneck, but can slow the performance of the IO subsystem.

Another important resource to monitor is CPU usage, which you can watch by using the following performance counters:

- **Processor: % Privileged Time** Percentage of time that the operating system spends processes SQL Server I/O requests. If this value is high at the same time that the disk-related performance counters described in the "Troubleshoot and analyze storage, IO, and cache issues" section are also high, your server likely needs a faster disk.
- **Processor: % Processor Time** Percentage of time that each processor spends executing a thread that is not idle. If this value is consistently between 80 and 90 percent, you should upgrade the CPU or add more processors.
- **Processor: % User Time** Percentage of time that the operating system executes user processes, including IO requests from SQL Server. If this value approaches 100%, it might indicate that the CPU is under pressure.
- **System: Processor Queue Length** Number of threads waiting for processor time. An increase in this counter means the CPU is not keeping up with demand and a faster processor is necessary.

To monitor memory, use the following performance counters:

- **Memory: Available Bytes** Amount of memory available for processes on the server. If this value is too low, check to see if another application on the server is failing to release memory or if memory on the server is adequate for your requirements.
- **Memory: Pages/sec** Frequency with which pages are retrieved from or written to disk due to hard page faults. When this value is consistently high, the server might be paging excessively. If it is, the Memory: Page Faults/sec performance counter will also be high.
- **Process: Working Set** Amount of memory used by a process. If this number is consistently lower than the minimum server memory option, SQL Server is configured to use too much memory.

Note Customizing the collection of performance counters

Refer to "Create a Data Collector Set to Monitor Performance Counters" at *https://technet.microsoft.com/en-us/library/cc722414.aspx* to customize a set of performance counters to collect and configure alerts. Step-by-step

instructions for saving performance counter data to SQL Server tables is available in "Capture PerfMon Statistics to SQL Server for Benchmarking, Baselining and Analysis," a blog post by David Ames at *http://www.amescode.com/capture-perfmon-statistics-to-sql-server-for-benchmarking-baselining-and-analysis/*.

SQL Trace

As we described in Skill 4.2, "Analyze and troubleshoot query plans," SQL Trace is useful for server-side or client-side tracing when you want to capture query plans. However, you can use tracing to monitor other aspects of SQL Server performance. Unlike DMOs that allow you to capture activity as it occurs, SQL Trace requires that a monitored event is completed before its data is added to the trace.

Important Considerations for query plan impact on performance

Remember that capturing a query plan does place some overhead on the server, so use with caution on a busy production server. In that case, the better approach is to use a server-side trace that stores output on a local drive. In addition, try to filter the trace as much as possible to limit the impact of tracing on your server.

Besides using SQL Trace to get query plans, consider using it as a diagnostic tool when you need to monitor SQL Server's behavior during query processing. For example, you might use SQL Trace in the following situations:

- **Lock escalation** You can find correlations between lock escalation events and queries running when lock escalation occurs by creating a trace that includes the Lock:Escalation, SP:Started, T-SQL:StmtStarted, SP:Completed, and T-SQL:StmtCompleted.
- **Deadlocks** As we described in Chapter 3, you can capture a deadlock graph by creating a trace that includes the Deadlock Graph, Lock:Deadlock, and Lock:Deadlock Chain events.
- **Slow queries** You can identify and capture information about slow queries in a trace containing the RPC:Completed, SP:StmtCompleted, SQL:BatchStarting, SQL:BatchCompleted, and Showplan XML events.

Note A resource for learning more about SQL Server Profiler

You can find in-depth information about using SQL Server Profiler, the graphical interface, in Brad M. McGehee's free ebook "Mastering SQL Server Profiler" at *http://www.red-gate.com/library/mastering-sql-server-profiler*.

Extended Events

We introduced Extended Events in Chapter 3 as a method for finding deadlocks after they occur and explored this feature again as lightweight alternative to SQL Trace for capturing query plans. The Extended Events feature in SQL Server provides even more functionality than deadlock and query plan monitoring.

As the replacement for SQL Trace, which is slated for deprecation in a future version of SQL Server, Extended Events not only allow you to perform the same tasks with greater flexibility and better performance, but also allow you to monitor more events. In SQL Server 2016, you can monitor 180 events by using SQL Trace, whereas you can monitor 1209 events by using Extended Events. Later in this chapter, we provide specific examples of how you might use Extended Events, we compare the performance impact between Extended Events and SQL Trace, and explain its architecture of this event-handling infrastructure.

Compare baseline metrics to observed metrics while troubleshooting performance issues

When using dynamic management objects to gain insight into SQL Server performance, you can only see current or recent historical information. Some problems become apparent only when you view how this information changes over time. Rather than wait for users to complain about application or query performance, you can proactively monitor the health of SQL Server by collecting baseline metrics and then periodically comparing current system behavior. That way you can identify negative trends as early as possible.

SQL Server includes a data collector to help you gather data and store it in the management data warehouse. This is a database that you use as centralized storage for many types of data, not just performance data. When you set up the management data warehouse, you specify which DMOs to query, which performance counters from the operating system and SQL Server to collect, and which SQL Trace events to capture. You can also use this infrastructure to capture other types of data that you might want to store centrally.

To set up management data warehouse in SQL Server Management Studio, expand the Management node in Object Explorer, right-click Data Collection, point to Tasks, and select Configure Management Data Warehouse. In the Configure Management Data Warehouse Wizard, click Next, and click New to open the New Database dialog box. Type a name for the database, click OK, and then click Next. On the Map Logins And Users page of the wizard, select the login for your SQL Server service account, select mdw_admin, and then click OK. Next click Finish, and then click Close.

To set up data collection, right-click Data Collection again, point to Tasks, and select Configure Data Collection. In the Configure Data Collection Wizard, click Next, and then select the server and database hosting the management data warehouse. Select the System

Data Collection Sets checkbox, click Next, click Finish, and then click Close.

To view information collected in the management data warehouse, right-click Data Collection once more, point to Reports, point to Management Data Warehouse, and select one of the following reports:

- **Server Activity History** This report, a portion of which is shown in Figure 4-43, displays data collected from DMVs and performance counters, such as waits, locks, latches, among other SQL Server statistics, and CPU, memory, disk, and network usage. By default, this information is gathered every 60 seconds, uploaded into the Management Data Warehouse every 15 minutes, and retained for 14 days. The report is interactive. When you click on a chart, a new report page displays. For example, when you click the SQL Server Waits report, you can view a table and chart containing wait statistics for the selected period of time.

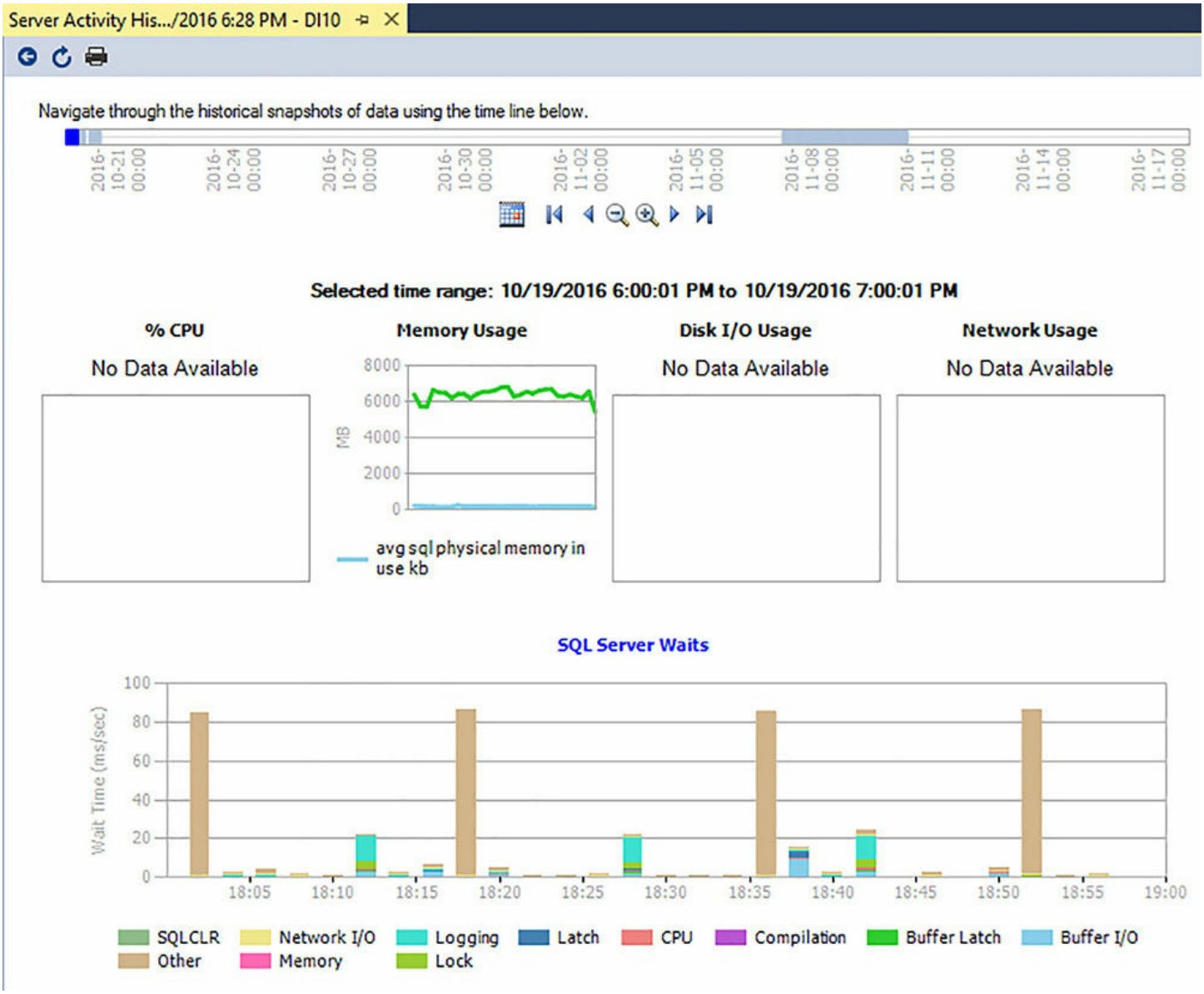
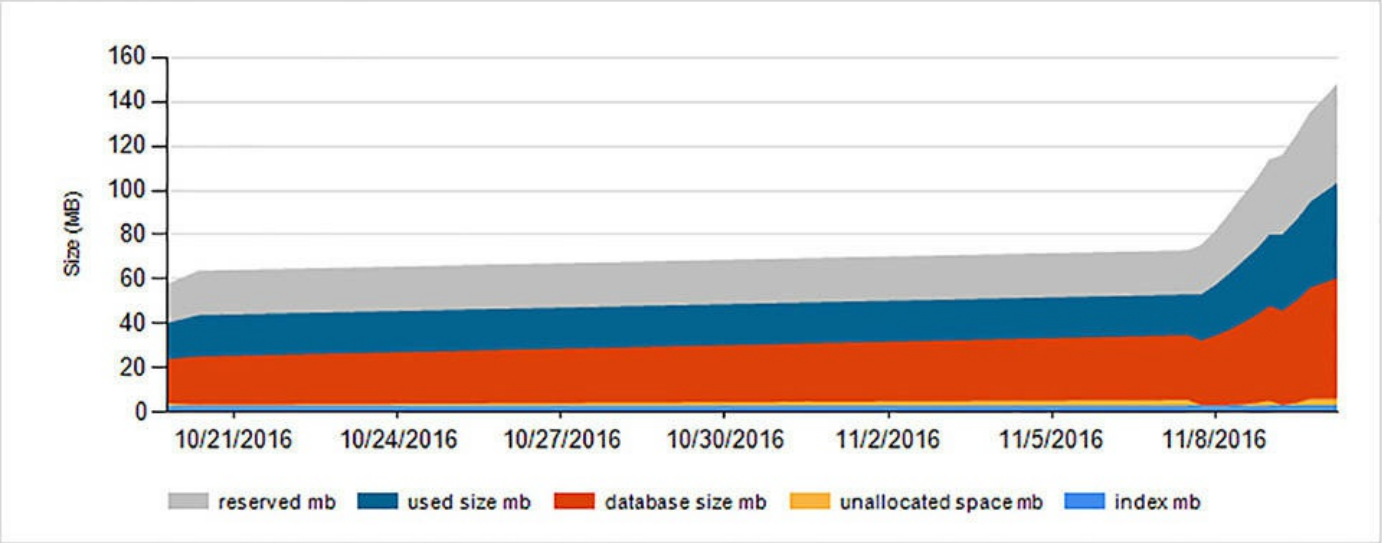


FIGURE 4-43 Server Activity History report

- **Disk Usage Summary** This report displays a table of databases listing the starting

and current size of the database and log files, the average growth measured as megabytes per day, and a sparkline chart to show the growth trend over time. You can click on a database name to view disk usage details for the database and log files as pie charts or click on the sparkline to view the disk space growth trends and collection details for each size metric, as shown in Figure 4-44.

This report provides the break down of the disk space used for on the server and growth trends for the database file for the last 12 collection points between 10/19/2016 5:55:41 PM and 11/10/2016 6:00:03 AM.



Collection Time	Database Size (MB)	Used Size (MB)	Reserved (MB)	Unallocated Size (MB)	Index Size (MB)
10/19/2016 5:55:41 PM	19.94	16.16	16.71	1.23	3.41
10/19/2016 6:00:02 PM	19.94	16.24	16.77	1.16	3.44
10/20/2016 8:15:28 AM	21.75	18.57	19.25	0.50	3.49
11/7/2016 12:01:00 PM	29.19	18.31	19.15	2.60	3.52
11/7/2016 6:00:36 PM	29.19	20.55	21.48	0.27	3.52

FIGURE 4-44 Disk Usage Summary subreport for a selected database

- **Query Statistics History** This report displays a column chart the top 10 queries within a specified interval by CPU usage, duration, total IO, physical reads, or logical writes and a table that includes the following query executions statistics: Executions/min, CPU ms/sec, Total Duration (sec), Physical Reads/sec, and Logical Writes/sec. The data is cached on the local file system and then uploaded to the Management Data Warehouse every 15 minutes.

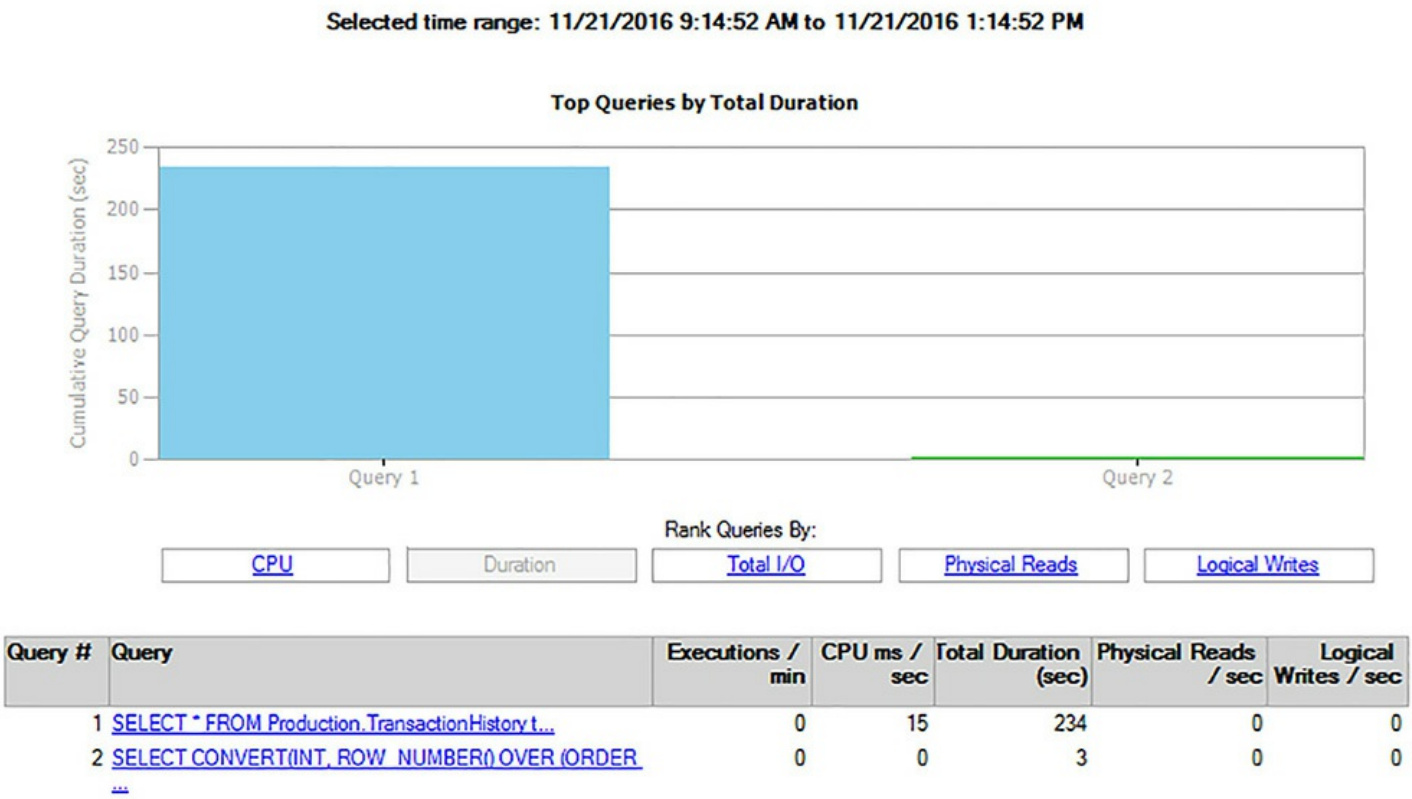


FIGURE 4-45 Query Statistics History report

You can click on one of these queries to view its details in a subreport, such as the query text, and more detailed execution statistics such as Average Duration (ms) per Execution or Average Executions Per Min. The subreport also displays a column chart of query plans that you can rank by CPU, duration, physical reads, or logical writes. Statistics by query plan are also available. Click on a query plan to view the query plan details much like the details for a query. Other links on this page allow you to view sampled waits or the graphical query plan.

Identify differences between performance monitoring and logging tools

Now that we have reviewed the various tools available for performance monitoring and logging, let's compare and contrast them to better understand their differences. That way, you can select the right tool for the particular task and objectives at hand. Table 4-1 provides a comparison of the tools by capability.

Capability	DMOs	Performance Monitor	SQL Trace	Extended Events	Management Data Warehouse
Requires coding	Yes	No	Yes, if using stored procedures	Yes, if using T-SQL statements	No
Graphical interface	No	Yes	Yes	Yes	Yes
Templates available	No	Yes	Yes	Yes	No
SQL performance monitoring	Yes	Yes	Yes	Yes	Yes
Operating system monitoring	Limited	Yes	No	Limited	Limited
Query plans	Yes	No	Yes	Yes	Yes
Deadlock graphs	Yes	No	Yes	Yes	No
Performance impact	Light	Light	Varies	Light	Light
Real-time activity	Yes	Yes	Yes	Yes	No
Historical activity	Only if saved to table	Yes	Yes	Yes	Yes
Save results to table	Yes	Yes	Yes	Yes	Yes
Alerting	No	Yes	No	No	No
Schedule at startup	No	Yes	No	Yes	No (data collection is scheduled at intervals)

TABLE 4-1 Comparison of performance monitoring and logging tools

Monitor Azure SQL Database performance

There are several diagnostic tools available to you for monitoring SQL Database performance. For a quick view of performance metrics, use the Azure portal. For more targeted analysis, you can use DMVs, Query Store, or Extended Events to monitor SQL Database performance just as you do for SQL Server.

Azure Portal

In the Azure portal, you can select the metrics that you want to monitor and display a chart that shows average utilization for selected metrics. To configure the chart, open the blade for your database in the Azure portal, and then click Edit inside the Monitoring chart. In the Edit Chart blade, select one of the following time ranges: Past Hour, Today, Past Week, or Custom. If you select Custom, you can type in a data range or use calendar controls to set a beginning and end date for the range. Next, select one of the following chart types: Bar or Line. Then select one or more of the following metrics:

- Blocked by firewall
- CPU percentage
- DTU limit
- DTU percentage
- DTU used
- Data IO percentage
- Database size percentage
- Deadlocks
- Failed connections
- In-memory OLTP storage percent
- Log IO percentage
- Sessions percentage
- Successful connections
- Total database size
- Workers percentage

After you add the metrics to monitor, you can view the results in the Monitoring chart in addition to details in the Metric window. When you select metrics, you can select compatible metrics only. For example, you can select metrics that count connections, as shown in Figure 4-46, or you can select metrics that measure the percentage of resources consumed, but you cannot mix those two types of metrics in the same chart.

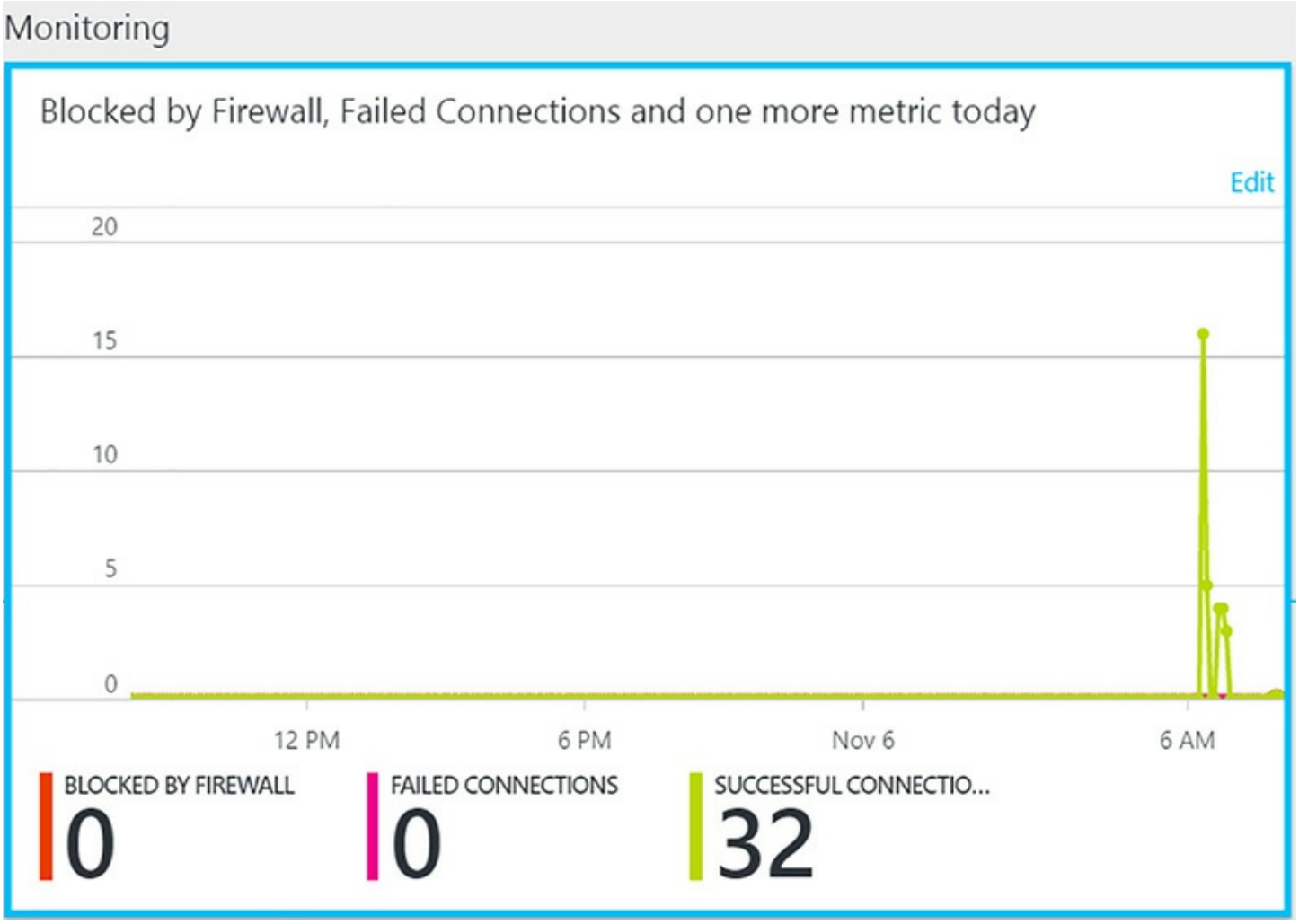


FIGURE 4-46 Monitoring chart for SQL Database in the Azure portal

You can also use the Azure portal to configure an email alert when a performance metric exceeds or drops below a specified threshold. Click the Monitoring chart to open the Metric blade. Then click Add Alert and follow the instructions in the Add Alert Rule blade. For example, you can add an alert to send an email when the number of connections blocked by the firewall exceeds 1, as shown in Figure 4-47. Not shown in this figure is the name of the resource, the name of the alert rule, and whether to email the alert to owners, contributors, and readers and optionally additional administrators.

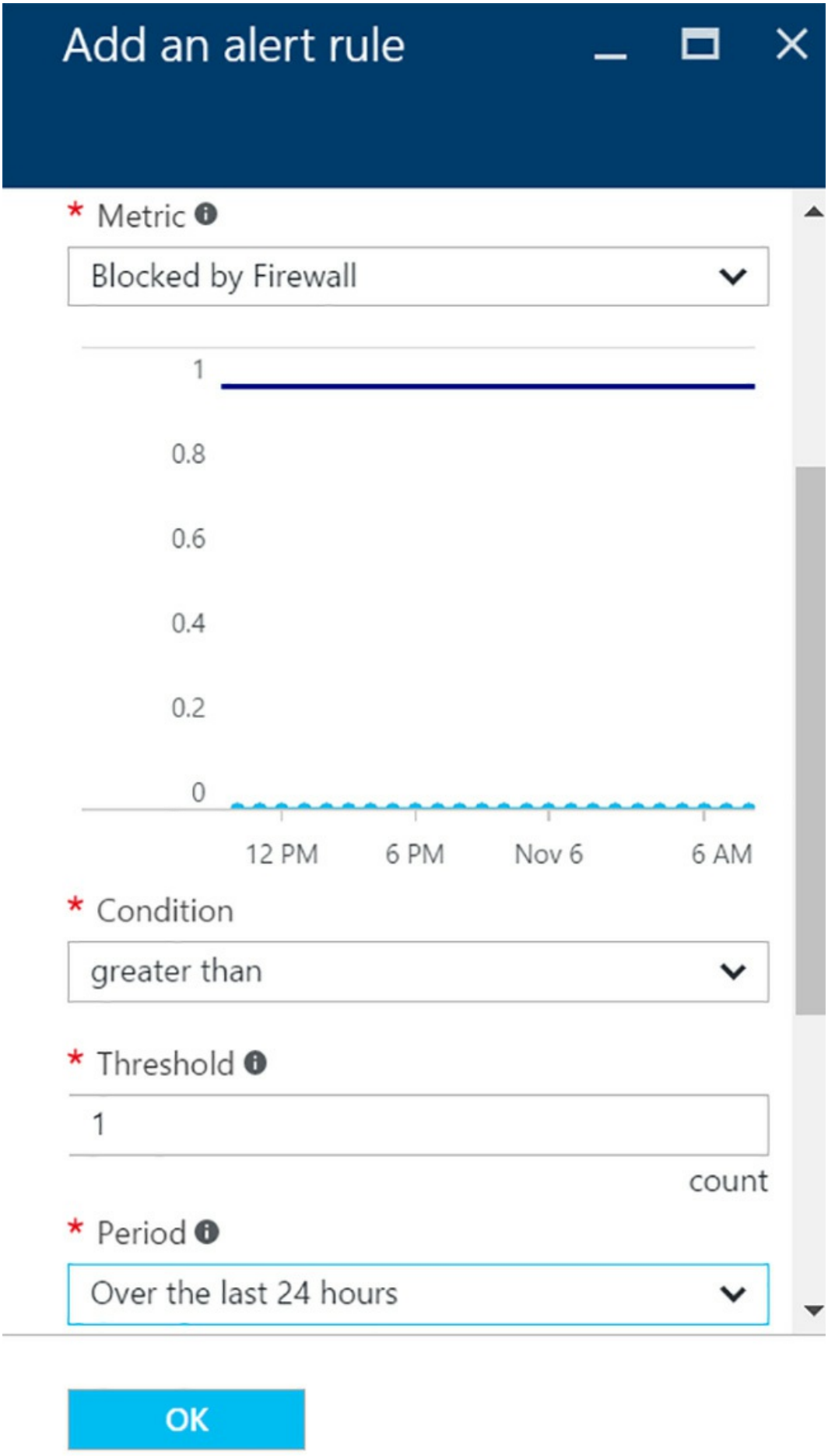

Add an alert rule
* Metric
Blocked by Firewall
1
0.8
0.6
0.4
0.2
0
12 PM
6 PM
Nov 6
6 AM
* Condition
greater than
* Threshold
1
count
* Period
Over the last 24 hours
OK

FIGURE 4-47 Add An Alert Rule blade for SQL Database

DMVs for SQL Database

Use the following DMVs to monitor SQL Database performance:

- **sys.database_connection_stats** Count successful and failed connections. The count of failed connections is the sum of login failures, terminated connections, and throttled connections.
- **sys.dm_db_resource_stats** Get the resource consumption percentages for CPU, data IO, and log IO. It returns one row for every 15 seconds, even when there is no activity in the database. For less granular data, you can use sys.resource_stats in the logical master database for your server.
- **sys.dm_exec_query_stats** In combination with sys.dm_exec_sql_text, find queries that use a lot of resources, such as CPU time or IO.
- **sys.dm_tran_locks** Discover blocked queries.
- **sys.event_log** Find issues such as deadlocking and throttling over the last 30 days. You must have permission to read the *master* database on the Azure server. As an example, you can search for specific types of events, such as deadlocks or throttle events, and when they occurred, as shown in Listing 4-42.

LISTING 4-42 Review SQL Server:Buffer Manager performance counters

Click here to view code image

```
SELECT
    Event_Category,
    Event_Type,
    Event_Subtype_Desc,
    Event_Count,
    Description,
    Start_Time
FROM sys.event_log
WHERE Event_Type = 'deadlock' OR
    Event_Type like 'throttling%'
```

> **Note Permission required for using DMVs in SQL Database**
>
> To use a DMV in SQL Database, you must be granted the VIEW DATABASE STATE permission.

Extended Events in SQL Database

You can use Extended Events to troubleshoot performance in SQL Database. It is similar to using Extended Events in SQL Server, although the set of extended events in SQL Database is smaller than the set available for SQL Server. In addition, there are some slight syntax differences when creating, altering, or dropping an event session as we noted in the "Monitor Azure SQL Database query plans" section earlier in this chapter.

Determine best practice use cases for extended events

Extended Events is a robust, yet lightweight tracing infrastructure that you can use to monitor and analyze SQL Server activity by collecting as much or as little information as you need. At minimum, you can use Extended Events for any diagnostic task that you can perform by using SQL Trace, but it can do so much more. In particular, Extended Events offers greater flexibility because you can filter events with more granularity. For this exam, you should be familiar with the following types of use cases that Extended Events supports:

- **System health** By default, an Extended Events session dedicated to system health information starts automatically when SQL Server starts. System health information includes session_id and sql_text for sessions with a severity greater than or equal to 20 or experiencing a memory-related error, non-yielding scheduler problems, deadlocks, long latch and lock waits, connectivity and security errors, and more. For complete details, see *https://msdn.microsoft.com/en-us/library/ff877955.aspx*.
- **Query performance diagnostics** Find historical deadlocks or queries that did not end, troubleshoot waits for a particular session or query, capture queries that match a specific pattern, and get query plans, to name a few. You can count the number of occurrences of a specific event to determine if a problem is recurring frequently.
- **Resource utilization monitoring and troubleshooting** You can create a session to capture information about server resources, such as CPU, IO or memory utilization. You can filter events for specific utilization thresholds to fine-tune the diagnostic process. Furthermore, you can correlate SQL Server events with Windows Event Tracing for Windows (ETW) logs that capture details about operating system activities.
- **Security audits** Capture login failures by filtering events with Severity 14 and Error 18456 with client application name to find malicious login attempts.

Distinguish between Extended Events targets

An Extended Event *target* receives information about an event. For example, in the "Capture query plans using extended events and traces" section earlier in this chapter, the target for the example Extended Events session is a file in which the session stores a query plan. In the last section of this chapter, we explain targets in greater detail. For now, be sure you understand the difference between the following targets for an Extended Events session:

- **etw_classic_sync_target** Unlike the other targets that receive data asynchronously, this is an ETW target that receives data synchronously. You use it to monitor system activity.
- **event_counter** This target counts the number of times that a specific event occurred.
- **event_file** This target writes the event session output to a file on disk in binary format. You use the sys.fn_xe_file_target_read_file function to read the contents of the file.
- **histogram** Like the event_counter target, the histogram target counts the occurrences of an event, but can count occurrences for multiple items separately and for both event fields or actions.
- **pair_matching** This target helps you find start events that do not have a corresponding end event. For example, you can discover when a lock_acquired event occurred without a matching lock_released event within a reasonable time.
- **ring_buffer** This target holds data in memory using an first-in, first-out method in which the oldest data is removed when the memory allocation is reached.

Compare the impact of Extended Events and SQL Trace

As we have explained throughout this chapter, you can use Extended Events and SQL Trace interchangeably to provide many similar diagnostic functions. However, it is important to note that there is considerable difference between them when considering their respective impact on the observed server. Both tools by necessity add overhead to the server, which is measurable by observing performance counters for monitoring CPU processor time and batch requests per second. The use of SQL Server Profiler for client-side tracing is the most intrusive option whereas using SQL Trace stored procedures for server-side tracing is less intrusive. With this in mind, you should try to limit the number of events and number of columns captured to minimize the overhead as much as possible.

The least intrusive option is Extended Events, which was developed as a lightweight replacement for SQL Trace. Nonetheless, because it does incur overhead, you should take care to create events that collect the minimum amount of data necessary for troubleshooting. In particular, be aware that the query_post_execution_showplan event is expensive and should be avoided on a production server. If you must use it to troubleshoot a specific issue, take care to restrict its use to a limited time only.

Define differences between Extended Events Packages, Targets, Actions, and Sessions

Throughout this chapter, we have discussed specific capabilities of Extended Events as well as the benefits of this feature, but we have yet to explore its components in detail. In this final section of the chapter, we review Extended Events concepts and objects that you should understand.

Events in Extended Events correspond to events in SQL Trace, but many more are supported in Extended Events to give you better diagnostic capabilities. An example of an event is sp_statement_starting.

Packages

A package is the top-level container for the various types of Extended Events objects: events, targets, actions, types, predicates, and maps. Of all the available packages, you can only use the following three packages in an event session:

- **package0** Contains Extended Events system objects
- **sqlserver** Contains objects related to SQL Server
- **sqlos** Contains SQLOS objects

Targets

A target is the destination for the data collected about an event. For short-term diagnostics, you can use a memory-resident target. To persist the data, you can use the event_file target. Review the section "Distinguish between Extended Events targets" earlier in this chapter to see the full list of available targets. After you create an Extended Event session, you can add one or more targets, like this:

Click here to view code image

```
ADD TARGET package0.event_file(SET
filename=N'C:\ExamBook762Ch4\query.xel',
    max_file_size=(5),max_rollover_files=(4)),
ADD TARGET package0.ring_buffer
```

Actions

An action is a response or series of responses that you bind to an event. For example, you can use an action to detect an execution plan or calculate run-time statistics. Or you can add information from the global state, such as session_id, to the firing event to aid in troubleshooting, like this:

Click here to view code image

```
ADD EVENT sqlserver.sql_statement_completed(
    ACTION (sqlserver.session_id,
        sqlserver.sql_text))
```

Sessions

An Extended Events session is the equivalent of a trace. When you create a session, you add an event and actions to fire with that event, define one or more targets for the data collected about the event, and optionally create predicates that define filters for the event. Listing 4-43 provides an example of an event session that captures session_id and sql_text

when a stored procedure executes on the server and sends the data to a file on the local drive and to the ring buffer.

LISTING 4-43 Create event session

Click here to view code image

```
CREATE EVENT SESSION [stored_proc]
ON SERVER
ADD EVENT sqlserver.sp_statement_completed(
    ACTION (sqlserver.session_id,
        sqlserver.sql_text))
ADD TARGET package0.event_file(SET
filename=N'C:\ExamBook762Ch4\query.xel',
    max_file_size=(5),max_rollover_files=(4)),
ADD TARGET package0.ring_buffer;
```

Note Learning more about Extended Events packages

There are other objects that can also exist in a package: types, predicates, and maps. However, these topics are not covered in the exam. If you'd like to learn more refer to "SQL Server Extended Events Packages" at *https://technet.microsoft.com/en-us/library/bb677278.aspx*.

Chapter summary

- SQL Server relies on statistics to select an optimal query plan. For this reason, it's important to keep statistics current. Statistics can become obsolete when a table is the target of high volume inserts or deletions because the data distribution can change significantly. You can use the DBCC SHOW_STATISTICS command to check the histogram and index densities for an index.
- SQL Server updates statistics automatically by default, but you can disable the automatic update and instead rely on a maintenance plan to update statistics when you need greater control over the timing of the update. When you create a maintenance plan, SQL Server creates a SQL Server Agent job that you can schedule to run at a convenient time.
- There are several DMOs available to help you manage indexes. Use sys.dm_db_index_usage_stats to review current index usage or, in combination with sys.indexes and sys.objects, to find indexes that are never used. Use sys.dm_db_index_physical_stats to find fragmented indexes. To find missing indexes, use sys.dm_db_missing_index_details, sys.dm_db_missing_index_groups, and sys.dm_db_missing_index_group_stats.

- The existence of overlapping indexes adds unnecessary overhead to SQL Server. You should periodically review index columns to ensure an index was not inadvertently added that overlaps with a similar index.
- Use the query_pre_execution_showplan or query_post_execution_showplan Extended Events as a lightweight method to capture the estimated or actual query plans, respectively. As an alternative, you can use SQL Trace system stored procedures for server-side tracing or use SQL Server Profiler to capture the Showplan XML or Showplan XML For Query Compile events for an estimated query plan or the Showplan XML Statistics Profile event for an actual query plan.
- When reviewing a query plan, you should review whether the following operators exist in the plan: Table Scan, Clustered Index Scan, Key Lookup (Clustered) or RID Lookup (Clustered) in conjunction with Index Seek (NonClustered), Sort, or Hash Match. Although the presence of these operators is not bad in all cases, you might consider options for tuning your query such as adding indexes to tables or adding filters to the query if operations are performed on large tables and your goal is to improve query performance.
- Query Store is a new feature in SQL Server 2016 and SQL Database that you can use to capture query plans and associated runtime execution statistics over time. Several built-in views are available to help you find queries that consume a lot of resources or have regressed from a previous point in time. You can also force a query plan when necessary.
- Usually an estimated query plan and an actual query plan are the same plan except that an actual query plan includes calculations and information that is available only after the query executes. You can create an estimated query plan to assess the impact of potential changes to a query or to table structures without executing the query. The actual query plan can vary from the estimated query plan if statistics are out-of-date or if the data structures changes after the estimated query plan is generated.
- Query Performance Insight is an Azure SQL Database features that graphically shows you which queries are consuming the most CPU, memory, IO, and DTU resources over time.
- The Resource Governor (available only in Enterprise, Developer, or Evaluation editions) allows you to define how SQL Server allocates CPU, memory, and IOPS resources on the server. You specify this allocation by creating one or more resource pools and then create workload groups that you assign to a resource pool. A workload group can have further resource consumption limits imposed and can be configured with relative importance within a resource pool to prioritize access to server resources when workload groups run concurrently. SQL Server assigns a session to a workload group by using criteria that you define in a classifier function. It if cannot match the session to a workload group, Resource Governor assigns the session to the default workload group assigned to the default resource pool.

- By using Elastic Scale for Azure SQL Database, you can manage your data in separate shards for which resource requirements can dynamically grow or shrink as needed to match the current workload. To use this feature, you must use the elastic database client library in your application to perform operations across all shards or in individual shards. You can also use the Split-Merge service to separate data in one shard into multiple shards or to combine data from multiple shards into a single shard.
- When choosing one of the following SQL Server 2016 editions, you must consider the scalability and features each supports: Express, Web, Standard, Enterprise, Developer, and Evaluation. For SQL Database, you choose the Basic, Standard, or Premium service level based on the maximum size of your database and the range of performance levels required. Performance levels are measured as DTUs which represent CPU, memory, and IO thresholds.
- One aspect of query performance is the efficiency of your storage and IO subsystems which you can optimize by managing file placement for system, data, and log files. Use filegroups as separate containers that can be spread across separate disks and optionally use secondary data files and partitioning to improve data access.
- The setup wizard for SQL Server 2016 makes it easier than in previous versions to optimize the configuration of tempdb by allowing you to define its file placement, the number of files to create, the initial file size, and autogrowth settings.
- As long as SQL Server is the only application running on a server, it can manage memory dynamically without intervention. When you must run other applications on the server, you can optimize SQL Server's memory configuration by specifying minimum and maximum memory thresholds, maximum worker threads, maximum memory for index creation, and minimum memory for query execution.
- You use the sys.dm_os_wait_stats, sys.dm_exec_session_wait_stats, or sys.dm_os_waiting_tasks DMVs to gather information about the amount of time that threads must wait on resources, determine whether the server is experiencing CPU, memory, or IO pressure, or find out which resources are causing excessive waits.
- You can troubleshoot IO issues by analyzing IO subsystem latencies captured in the sys.dm_io_virtual_file_stats and sys.master_files DMVs or by reviewing performance counters for SQL Server's buffer manager in the sys.dm_os_performance_counters DMV.
- You can troubleshoot cache issues by using the following DMVs: sys.dm_os_memory_cache_counters, sys.dm_os_sys_memory, or sys.dm_os_memory_clerks. You can also use performance counters for SQL Server's buffer manager and memory manager in the sys.dm_os_performance_counters DMV.
- SQL Database query plans are accessible by using Showplan SET options, Extended Events, or Query Store, but not by using SQL Trace.

- You have a variety of tools that you can use to monitor operating system and SQL Server performance metrics: DMOs, Performance Monitor, SQL Trace, and Extended Events. In many cases, you are likely to use a combination of tools to diagnose SQL Server behavior.
- Management Data Warehouse is a SQL Server feature that allows you to capture performance-related information over time in contrast to other available monitoring tools which primarily provide point-in-time or cumulative information. Management Data Warehouse provides the following reports for your review: Server Activity History, Disk Usage Summary, and Query Statistics History.
- To monitor SQL Database performance, you can monitor specific metrics in the Azure portal. In addition, you can use the following DMVs with SQL Database for monitoring: sys.database_connection_stats, sys.dm_db_resource_stats, sys.dm_exec_query_stats, sys.dm_tran_locks, and sys.event_log. Extended Events are also available for SQL Database, although the set of supported events is smaller than the set that supports SQL Server.
- You should be familiar with the following best practice use cases for extend events: system health, query performance diagnostics, resource utilization monitoring and troubleshooting, and security audits.
- You should also understand how Extended Events uses the following targets to store information about an event: etw_classic_sync_target, event_counter, event_file, histogram, pair_matching, and ring_buffer.
- Although Extended Events and SQL Trace can often be used interchangeably, Extended Events has a much lower impact on performance. You can measure the difference in impact by observing CPU processor time and batch requests per second.
- The Extended Events architecture is comprised of several components. A package is the top-level container for the other objects. As data is collected about an event, it is sent to a target. In addition, you can configure an action to occur in response to an event. A session is a set of events, actions, and targets that you configure and enable or disable as a group.

Thought experiment

In this thought experiment, demonstrate your skills and knowledge of the topics covered in this chapter. You can find answer to this thought experiment in the next section.

You recently started as a new database administrator at Consolidated Messenger. Because you are unfamiliar with this SQL Server environment as a new employee, you decide to perform some analysis to determine if there are any significant problems to fix.

1. Which query or command do you use to find the most recent update for statistics on tables or indexed views?

A. DBCC SHOW_STATISTICS('ConsolidatedMessengerDB', All Indexes);

B. SELECT name, STATS_DATE(object_id, stats_id) FROM sys.stats WHERE object_id IN (SELECT object_id FROM sys.objects WHERE type = 'U');

C. SELECT name, auto_created (object_id, stats_id) FROM sys.stats WHERE object_id IN (SELECT object_id FROM sys.objects WHERE type = 'U');

D. SELECT name, auto_created (object_id, stats_id) FROM sys.stats WHERE object_id IN (SELECT object_id FROM sys.objects WHERE type = 'U');

2. When you query the sys.dm_db_index_physical_stats DMV, you see the output shown below. What problem do you see and what step should you take to resolve it?

Click here to view code image

```
idx_id
ObjectName                                    index_type_desc
AvgPageSpacePct     frag_ct  AvgFragPct
------- ------------------------------------ ------------
------------------
------ ------------------------- ---------- -------------
-------------
1          PK_SalesOrders_OrderD        CLUSTERED
INDEX           2037
54.9851742031134 237        34.02189781021898
2          IX_SalesOrders_CustomerID NONCLUSTERED
INDEX   685    98.4313442055844
2             0
```

3. You enabled Query Store on the main corporate database several weeks ago. Which DMV do you use to locate the top 5 queries with the longest duration?

A. sys.query_store_plan

B. sys.query_store_query

C. sys.query_store_query_text

D. sys.query_store_runtime_stats

E. sys.query_store_runtime_stats_interval

4. When you use Query Store to examine the query plans for a frequently executed query, you notice that for one plan uses an Index Scan operator and a second plan uses and Index Seek operator. If the normal query pattern is to retrieve a small number of rows, which is the more optimal query plan and how can you require SQL Server to use it?

5. You have been monitoring wait statistics in the sys.dm_os_wait_stats DMV for several weeks and notice that the ratio of signal_wait_time_ms to wait_time_ms has been increased from 10% to 30%. What type of problem is this likely to indicate?

A. CPU pressure

B. Memory pressure

C. Network bandwidth issues

D. IO subsystem failures

6. As another method to confirm your diagnosis for the scenario in the previous question, which performance counters should you check?

   A. Physical disk counters: % Disk Time, Avg. Disk sec/Read, Avg. Disk sec/Transfer, Avg. Disk sec/Write, Current Disk Queue Length.

   B. Processor counters: % Privileged Time, % Processor Time, % User Time, Processor Queue Length.

   C. Memory counters: Available bytes, Pages/sec, Working set.

   D. SQL Server counters: Page lookups/sec, Page reads/sec, Page writes/sec, Free List Stalls/Sec, Lazy Writes/Sec, Memory Grants Outstanding, Memory Grants Pending.

7. Which tool monitoring tools can you use to get information about SQL Server memory usage without writing any code?

   A. DMVs or SQL Profiler

   B. Server-side SQL Trace or Extended Events

   C. Performance Monitor or Management Data Warehouse

   D. Client-side SQL Trace or Resource Governor

Thought experiment answer

This section contains the solution to the thought experiment.

1. **The answer is B**. The sys.stats catalog view contains both the stats_id and object_id columns necessary to use the STATS_DATE system function that returns the most recent update date for an object's statistics. The `DBCC SHOW_STATISTICS` command requires you to include a specific index name as the second argument, therefore A is incorrect due to the syntax. Similarly, C and D are examples of incorrect syntax because T-SQL does not include an `auto_created` function.
2. The clustered index has a page count greater than 500 and fragmentation is 34%. In this case, you should rebuild the index.
3. **The answer is D**. You can order by the avg_duration column in descending order in the sys.query_store_runtime_stats DMV to find the queries that run the longest. The sys.query_store_plan in A is incorrect because this DMV includes only information about estimated query plans. Sys.query_store_query is incorrect in B because this DMV collects aggregated information for a query's compilation, but does not collect execution statistics. On its own the sys.query_store_query_text DMV in C is not a

correct answer because it includes the query text without duration information, although you can join it to sys.query_store_runtime_stats to get more complete information about long-running queries. Sys.query_store_runtime_stats_interval is stores information about the intervals of time during which statistics are captured, but does not report duration information.

4. When multiple query plans exist for a selected query, Query Store allows you to select a query plan and then force that plan for all subsequent executions of the same query. In this example, the more optimal query plan is the one that includes the Index Seek because it incurs a lower cost by selecting specific rows from the index rather than scanning the entire index to find the rows to select.

5. **The answer is A**. The signal wait time is the amount of time that a thread is able to run a task, but is waiting its turn on the CPU. To confirm this diagnosis, check for an increasing number of SOS_SCHEDULER_YIELD wait types in the sys.dm_os_wait_stats DMV and check the sys.dm_os_schedulers DMV for a high value in the runnable_tasks_count column. Answer B is incorrect because memory pressure is indicated by PAGEIOLATCH waits in combination with a Page Life Expectancy performance counter value dropping over time. Answer C is incorrect because you generally use network-related wait types and performance counters to confirm your diagnosis of network bandwidth issues. Answer D is incorrect because a failure in the IO subsystem will become evident when there are many waits containing IO in the name and average wait time begins to increase.

6. **The answer is B**. You can monitor CPU usage by using the Processor performance counters. In particular, if % Processor Time is between 80 and 90 percent consistently, % User Time nears 100% consistently, and the Processor Queue Length value is increasing over time, the CPU is experiencing pressure. You should consider upgrading the CPU or adding more processors. The counters listed for Answers A, C, and D are useful performance counters. However, they do not provide any confirmation of whether the observations in Question 5 are related to CPU pressure.

7. **The answer is C**. Both Performance Monitor and Management Data Warehouse provide graphical interfaces that you can use to monitor SQL Server's memory usage. In Performance Monitor, you can set up the data collection of performance counters for the SQL Server Memory Manager, such as Memory Grants Outstanding or Memory Grants Pending, among others. In Management Data Warehouse, you can use the Server Activity History report to drill into Memory Usage details for the server. To access information from DMVs, you must write a T-SQL query; therefore, A is incorrect. Similarly, B is incorrect because a server-side SQL Trace requires you to write code. Answer D is incorrect because Resource Governor is not a performance monitoring tool.

Index

A

abbreviations 11
ACID properties 196–203
actual query plans 314–319
AFTER triggers 160, 176–179
algebrizer 283
ALTER INDEX statement 89, 90, 92
ALTER statement 61
ALTER TABLE command 120–122
ALTER TABLE statement 14–15, 27, 253
approximate numeric data type 16
artificial keys 5, 47
ASCII characters 18
atomic blocks 249
atomicity 196, 198–201
attributes 3
  relationship of non-key to key 8–11
auto-commit transactions 203
autocommit transactions 151
AUTO_CREATE_STATISTICS 271
Autogrowth 338
AUTO_UPDATE_STATISTICS 271
AUTO_UPDATE_STATISTICS_ASYNC 271
Azure Access Panel. *See* Access Panel
Azure portal 319–320, 356–357
Azure SQL Database
  DMVs for 358
  Extended Events in 358
  performance monitoring 356–358
Azure SQL Database Performance Insight 283, 319–324
Azure SQL Database query plans 346

B

backward compatibility 64

baseline performance metrics 347–362
  compared to observed metrics 352–355
batch execution mode 79–80
batch separator 12
BEGIN ATOMIC_END clause 249
BEGIN TRANSACTION statement 151, 205, 207
binary data 17
bookmark lookups 30
bottlenecks 242, 243, 340, 345
Boyce-Codd normal form 9
brackets 11
B-Tree indexes 24–50
  designing 26–40
  structure 24–26
bulk data loading
  into clustered columnstore index 89–93
bulk update locks 214
business requirements
  for database 2–4
  view structure design based on 53–57
business rules
  enforcing with constraints 102–115

C

cache issues 345–346
Camel Casing 12
cardinality 74, 266–270
cascading operations 113–115
CASE statement 247
character data 17
CHECK constraint 65
CHECK constraints 102, 107–111, 121, 160
classification process 325
classifier user-defined functions 329–330
client-side tracing 290–292, 360
CLR. *See* Common Language Runtime (CLR)
CLR stored procedures 130

clustered columnstore indexes 73–80, 85–88
  bulk data loading 89–93
clustered indexes 25, 46–48, 69, 85
Clustered Index Scan operator 36, 50, 69, 293–294
clustering key 46–47, 48
COALESCE() function 20
columnar data 165–167
columnar databases 70–71
column constraints 102
COLUMNPROPERTYEX() function 34
columns 5, 5–6
  adding 14–15
  cardinality of 266–270
  clustered index 46–48
  computed 20–21, 34
  data format for 109–110
  data types for 15–24
  density of 268
  dropping 15
  foreign key 28–32, 74
  indexed vs. included 41–45
  limiting to set of values 118–119
  NOT NULL 104
  NULL values in 28, 106
column segments
  in columnstore indexes 72
columnstore indexes 70–93, 252
  adding rows to compressed rowgroups 83
  attributes of 85
  batch execution mode 79–80
  clustered 73–80, 85–88
  loading data into 89–93
  maintenance 89–95
  non-bulk operations on 93–95
  non-clustered 78, 80–88
  structure of 71–72
  targeting analytically valuable columns in 83

use cases 70–73
Columnstore Scan operator 88
COMMIT statement 204–205
COMMIT TRANSACTION 151
COMMIT TRANSACTION statement 205
Common Language Runtime (CLR) 130
common table expressions (CTEs) 116, 248
company names 18–19
comparison operators 59
composite indexes 25, 39
compressed rowgroups 83
COMPRESSION_DELAY setting 83
computed columns 20–21
indexing 34
Compute Scalar operator 30
concurrency 26
concurrent processes
potential problems with 211–216
concurrent queries
results of 219–228
connectors. *See also* receive connectors; *See also* send connectors
consistency 196, 201
constraints 14, 61, 101, 102–129
adding to tables 119–122
CHECK 65, 102, 107–111, 121, 160
column 102
DEFAULT 48, 65, 102, 103–105
FOREIGN KEY 73, 75, 102
relating to UNIQUE constraint 117–119
use of 110–116
PRIMARY KEY 27, 28, 41, 47, 65, 102, 106, 111–112, 120, 125–130
results of DML statements and 123–125
table 102
UNIQUE 102, 105–106, 117–119, 120
uniqueness 27–28
covering indexes 41–45, 57, 77
CPU pressure 341

CPU usage 350
CREATE COLUMNSTORE INDEX statement 83
CREATE INDEX statement 29, 39, 43, 48
CREATE SCHEMA statement 12
CREATE TABLE statement 12–15
CTEs. *See* common table expressions (CTEs)

D

DAC. *See* Datacenter Activation Coordination
DAGs. *See* Database Availability Groups
DAS. *See* Direct-Attached Storage
data
  binary 17
  character (string) 17
  hiding 53
  limiting, through DDL 61–62
  modifying in views, with multiple tables 62–64
  redundant 8, 9
  reformatting 53, 54–55
  view structure design to select 53–57
data access
  real-time 244
database concurrency 195–264
  isolation levels and 216–229
  locking behavior and 230–241
  memory-optimized tables and 242–255
  natively compiled stored procedures and 242, 255–258
  transactions and 195–216
DATABASE DDL triggers 172–174
database design
  adding indexes during 26–32
  based on business requirements 2–4
  denormalization 11
  determining data types 15–24
  using normalization 4–11
  writing table create statements 11–15
Database Engine Tuning Advisor 281

database file optimization 336–337
database instances
  performance management for 324–346
    database file optimization 336–337
    database workload 325–331
    Elastic Scale 331–333
    memory optimization 339–340
    query plans 346
    storage, IO, and cache troubleshooting 343–346
    wait statistics 340–343
  tempdb optimization 338–339
Database Manipulation Language (DML)
  statements 196–203
database objects 1–100
  changing definition of 61
  columnstore indexes 70–93
  dropping 61
  indexes 24–50
  naming 11–12, 12–13
  relational database schema 2–24
  views 52–69
databases. *See* mailbox databases
  columnar 70–71
  logical database model 4
  tuning 32
database scalability 331–333
Database Throughput Unit (DTU) 321–322
database workload
  managing, in SQL Server 325–331
  Resource Governor management queries 330–331
  resource pools 326–328
  user-defined functions 328–329
  workload groups 328
Databse Definition Language (DDL) triggers 159
Data Definition Language (DDL) 123
Data Definition Language (DDL) triggers 169–174
data distribution 266–270

data files 336
data format 107, 109–110
data input
  CHECK constraints on 107–111
  coordinating multiple values 110–111
data integrity
  complex 160–164
  constraints for 102–129
data loading
  into columnstore index 89–95
Data Manipulation Language (DML) 123–125
Data Manipulation Language (DML) triggers 159–160, 169. *See also* triggers
data storage. *See also* storage architectures; *See also* storage requirements
datatype
  choosing 108–109
  conversion 144
  int 107
  precedence 144
data types 15–24
  approximate numeric 16
  binary data 17
  character (string) data 17
  computed columns 20–21
  considerations for choosing 18–20
  date and time values 16–17
  dynamic data masking 20, 21–24
  for clustering key 48
  importance of choosing 15–16
  not supported in columnstore indexes 71
  other 17–18
  precise numeric 16
data values 15
data warehouses
  clustered columnstore indexes and 73–80
date values 16–17
DBCC SHOW_STATISTICS command 266–268
DDL. *See* Data Definition Language (DDL)

deadlock graphs
capture and analyze 237–240
Extended Events 240
SQL Server Profiler 239–240
deadlocks 351
remediation of 241–242
decimals 16
DEFAULT constraint 48, 65
DEFAULT constraints 102, 103–105
DEFAULT keyword 182
DEFAULT VALUES 104–105
degenerate dimensions 75
degree of parallelism (DOP) 328
delayed durability 254
delayed durable transactions 202–203
DELETE statement 63
delimiters 11, 12
deltastore structure 72
denormalization 11
design
database
adding indexes during 26–32
based on business requirements 2–4
determining data types 15–24
using normalization 4–11
writing table create statements 11–15
indexes 26–40
tables
improving, using normalization 4–11
deterministic calculations 20
deterministic functions 186–188
Developer edition 335
DFS. *See* Distributed File Share
dimensional formatted data warehouses
using clustered columnstore indexes on 73–80
dimension keys 74
dimensions 74

dirty reads 212
Disk Usage Summary report 353–354
distributed transactions 208–209, 211, 219
divide-by-zero errors 180–181
DML. *See* Data Manipulation Language (DML)
DMV. *See* dynamic management view
DMVs. *See* dynamic management views (DMVs)
doomed transactions 152
double-quotes 11
DROP command 61
DROP [objectType] IF EXISTS command 61
duplicate key values 46
durability 197, 202–203
durable memory-optimized tables 254
dynamic data masking 20, 21–24
dynamic link library (DLL) files 246, 247
dynamic management objects (DMOs) 276–281, 340–343, 347, 348, 355
dynamic management view (DMV) 45
dynamic management views (DMVs) 231–237, 256–259, 276–280, 304–305, 341–342, 345–346, 358

E

EFS. *See* Encrypting File System
elastic database client library 332
Elastic Scale 331–333
email() function 23
ENCRYPTION 52
Enterprise edition 334–335
error handling
  @@ERROR system function for 148–149
  in stored procedures 144–158
  rethrowing errors 150
  throwing errors 144, 145–147
  THROW statement for 157
  transaction control logic in 144, 151–158
  TRY...CATCH construct for 149–151, 153–154, 157
ERROR_PROCEDURE() function 158

@@ERROR system function 148–149, 154
ESRA. *See* EdgeSync replication account (ESRA)
estimated query plans 268–270, 314–319
ETDATE() function 20
Evaluation edition 335
EVENTDATA() function 171
exclusive locks 213, 229, 236
EXECUTE AS statement 24
EXECUTE statement 142
execution statistics 256–259
explicit transactions 151, 203–204, 205–209
Express edition 334
Extended Events 283–286, 313, 346, 348, 352, 355, 358
  actions 360
  best practice use cases for 359
  compared with SQL Trace 360
  events 360
  packages 360
  sessions 360
  targets 359–360, 360
Extended Events deadlock graph 240
external resource pools 327
Extract-Transform-Load (ETL) operations 243, 245–246

F

fact tables 74, 78
federated SQL Servers 64
file groups 336–337
file placement 336, 338
file size 338
filtered rowstore indexes 29, 84
first normal form 6–7
forced query plans 313–314
foreign key columns 28–32, 74
FOREIGN KEY constraints 28–32, 75, 102
  cascading operations 113–115
  limiting column to set of values using 118–119

PRIMARY KEY constraints and 111–112
relating to UNIQUE constraint 117–119
table hierarchies and 115–116
use of 110–116
forms 4
FROM clause 60, 248
FSW. *See* File Share Witness
FUNCTION query 182
functions
deterministic 186–188
non-deterministic 186–188
system. *See* system functions
user-defined. *See* User-Defined Functions (UDFs)

G

globally unique identifier (GUID) 18, 48
Globally Unique Identifiers (GUIDs) 128
GO 12
GROUP BY query 182
GUID. *See* globally unique identifier

H

hash indexes 252
Hash Match (Aggregate) operator 298–299
Hash Match (Inner Join) operator 299–302
Hash Match join operator 51
Hash Match operator 35, 37, 51, 77
heaps 25
heirarchyId data type 18
high-concurrency databases
transactions in 211–216
hygiene. *See* message hygiene

I

IDENTITY property 65, 104, 105, 126–128
implicit transactions 203–205, 216
included columns 41–45

INCLUDE keyword 43, 46
index create memory 340
indexed columns 41–45
indexed views 67–69
indexes 24–50, 252–253
  adding during coding phase 26
  adding during database design phase 26–32
  bookmark lookups 30
  clustered 25, 46–48, 69, 85
  columnstore 70–93
    clustered and non-clustered 73–87
    data loading 89–95
    maintenance 89–95
    use cases 70–73
  composite 25, 39
  consolidating overlapping 281–282
  covering 41–45, 57, 77
  designing 26–40
    common search paths 32–35
    foreign key columns 28–32
    once data is in tables 32–40
    uniqueness constraints 27–28
  filtered 29, 84
  finding unused 278–279
  fragmented 279–280
  identifying missing 280–281
  indexed vs. included columns 41–45
  joins 35–37
  missing 45, 46
  non-clustered 25, 85
  optimization of 266–284
  optimizing 28
  query plans 30–32, 34–36, 42, 46, 49–52
  review current usage 276–279
  simple 25, 39
  sorts 38–41
  structure 24–26

uses of 24, 31–32
using dynamic managemet objects to review 276–281
index keys 42
maximum size of 25–26
Index Seek (NonClustered) operator 294–295
In-Memory OLTP feature 242
inner data set 301
INSERT INTO clause 105
INSERT statements 248
INSTEAD OF INSERT triggers 168
INSTEAD OF triggers 160, 166–170, 176–179
INSTEAD OF UPDATE triggers 166
int data type 107
integers 16, 48
intent locks 213–214
internal resource pools 327
Internet of Things 243
interpreted SQL stored procedures 130
interpreted stored procedures 249
IO issues 341
IO troubleshooting 343–345
isolation 196–197, 201–202
isolation levels 216–229
concurrent queries based on 219–228
differences between 217–219
READ COMMITTED 218, 220–221, 229
READ_COMMITTED_SNAPSHOT 219, 227–228, 230–231
READ UNCOMMITTED 218, 221–222, 229
REPEATABLE READ 218, 222–223, 229
resource and performance impact of 228–230
SERIALIZABLE 218, 223–224, 229
SNAPSHOT 219, 224–227, 229

J

joins 35–37

K

key attributes
relationship to non-key attributes 8–11
Key Lookup (Clustered) operator 294–295
Key Lookup operator 38
key-range locks 214–215
keys 5, 6
artificial 5
natural 6
surrogate 6

L

lazy commits 202–203
linked servers 67
lock compatibility 215
lock escalation events 351
lock hierarchy 213–215
locking behavior 230–241
deadlocking 237–241
escalation behaviors 237–238
troubleshooting 231–237
log files 336. *See* transaction log files
logging tools 355
login triggers 159, 174–176
logon triggers 169, 174–176
log sequence numbers (LSNs) 202
long running queries 323–324
lost updates 212
Lync Online. *See* Skype for Business

M

maintenance
columnstore indexes 89–95
Management Data Warehouse 352–355
materialized views. *See* indexed views
max server memory 339
max worker threads 340
measures 75

memory cache 345–346
memory optimization 339–340
memory-optimized tables 242–255
  analytics workloads 253
  durability options 253–254
  indexes 252–253
  natively compile stored procedures and 247–252
  performance optimization of 245–255
  SQL Server editions supporting 245
  use cases for 242–245
memory pressure 342
memory usage monitoring 350
Merge Join operator 38, 40–41
MERGE statement 248
message transport. *See* transport
Microsoft Azure. *See* Azure
Microsoft Azure Active Directory. *See* Azure Active Directory (Azure AD)
min memory per query 340
min server memory 339
missing indexes 45, 46
monetary values 16, 19

N

names
  company 18–19
  database objects 12–13
  object 11–12
NAT. *See* network address translation (NAT)
natively compiled objects 130
natively compiled stored procedures 242
  creating 247–252
  execution statistics for 256–259
  unsupported T-SQL constructs for 249
  usage scenarios for 255
natural keys 6
Nested Loop operator 78
nested loops 35

nested transactions 206–207
NEWSEQUENTIALID() function 128
NICs. *See* network interface cards (NICs)
non-bulk operations
  on columnstore 93–95
nonclustered B-tree indexes 252
non-clustered columnstore indexes 73, 78, 80–88
  designing in conjuction with clustered 85–88
  filtered 84–85
  on OLTP tables 81–85
non-clustered indexes 25, 85
non-deterministic functions 186–188
non-durable memory-optimized tables 254
non-key attributes
  relationship to key attributes 8–11
non-repeatable reads 212
normalization
  Boyce-Codd normal form 9
  defined 4
  first normal form 6–7
  rules
    covering relationship of non-key attributes to attributes 8–11
    covering shape of table 5–7
  second normal form 8–10
  table design using 4–11
  third normal form 8, 9–11
NOT NULL columns 104
NULL columns 106
NULL expression 13
NULL values 28, 29

O

object naming 11–13
objects 3
Office Telemetry. *See* telemetry
OLTP. *See* online transaction processing
OLTP databases 28

OLTP tables
  using non-clustered columnstore indexes on 81–85
ON clause 14
one-to-one cardinality 10
online transaction processing (OLTP) 1–100
operating system performance metrics 347–352
ORDER BY clause 38, 40, 52
outer data set 301
Overall Resource Consumption view 314

P

parallelism 76
parameters
  input and output 135–137
  stored procedures 132, 135–139
  table-valued 137–139, 244
  type mismatch 143–144
parameter sniffing 310
partial() function 23
partitioned views 64–67
partitioning 14, 337–338
Pascal-casing 12
PERCENT clause 248
performance
  data types and 15
performance counters 344–346, 348–351
Performance Monitor 347, 348–351, 355
performance monitoring
  Azure SQL Database 356–358
  baseline performance metrics 347–362
  DMOs for 276–280, 340–343, 347, 348, 355
  Extended Events for 355, 358, 359–362
  Management Data Warehouse 352–355
  Performance Monitor for 347, 348–351, 355
  SQL Trace for 347, 355
  vs. logging 355
phantom reads 212

plan summary 308, 310, 311
platform-as-a-service. *See* PaaS
precise numeric data type 16
prefixes 11
PRIMARY KEY constraint 14, 27, 28, 41, 47, 65
PRIMARY KEY constraints 102, 106, 120
  FOREIGN KEY constraints and 111–112
  use of 125–130

Q

quantum 340
queries
  concurrent 219–228
  dimensional data warehouses 73–80
  grouping data in 76
  long running 323–324
  parallelism 76
  range 47
  resource consuming 321–323
  Resource Governor management 330–331
  slow 351
  statistics collection 258–259
  that return large results 47
Query Menu 31
query optimizer 283
query parsing 283
Query Performance Insight 319–324
query plan operators 291–301
  Clustered Index Scan operator 293–294
  Hash Match (Aggregate) operator 298–299
  Hash Match (Inner Join) operator 299–302
  Hash Match operator 296
  Index Seek (NonClustered) operator 294–295
  Key Lookup (Clustered) operator 294–295
  Sort operator 295–297
  Table Scan operator 293
query plans 30–32, 34–36, 35–36, 42, 46, 49–52, 268–270, 283–324

Azure SQL Database 346
Azure SQL Database Performance Insight 319–324
capturing, using extended events and traces 283–292
comparing estimated and actual 314–319
estimated 268
forced 313–314
performance impacts of 351
poorly performing operators 291–301
query_post_execution_showplan 284
query_pre_execution_showplan 284
Query Statistics History report 354
Query Store 283, 346
components 304–306
enabling 319–320
properties 302–304
views 306–314
query_store_plan_forcing_failed Extended Event 313

R

RAISERROR statement 145–147
range queries 47
READ COMMITTED isolation level 218, 220–221, 229
READ_COMMITTED_SNAPSHOT isolation level 219, 227–228, 230–231
READ UNCOMMITTED isolation level 218, 221–222, 229
real data types 19
real-time analytics
using non-clustered columnstore indexes 81–85
real-time data access 244
REBUILD setting 89, 92
redundant data 8, 9
Regressed Queries view 314
relational database schema 2–24
designing
based on business requirements 2–4
determining data types 15–24
using normalization 4–11
writing table create statements 11–15

REORGANIZE setting 89, 90, 92
REPEATABLE_READ isolation level 213, 218, 222–223, 229
reporting
  views for 53, 55–56
resource consuming queries 321–323
resource consumption monitoring 330–331, 359
Resource Governor 325–331
  management queries 330–331
resource locks 212–216
resource pools 325, 326–328
RETURN statement 139–140
REVERT statement 24
ROLLBACK TRANSACTION statement 151, 164, 207, 209
row groups 83
  in columnstore index 71
Row-Level Security feature 53
rows 5, 5–6
rowstore 24
rowstore indexes
  uses of 85
rowversion data type 18

S

savepoints 157, 209–211
SAVE TRANSACTION statement 209
scalar user-defined functions 180–183
SCHEMABINDING 52
SCHEMABINDING clause 249
schema locks 214
schema modification locks 236
schemas
  defined 3
  designing
    based on business requirements 2–4
  relational database 2–24
SCOPE_IDENTITY() function 136
search conditions 35

secondary data files 336–337
secondary uniqueness criteria 105–106
second normal form 8–10
security
  Row-Level Security feature 53
security audits 359
SELECT clause 183
SELECT INTO clause 248
SELECT statement 317
SELECT statements 52
self-service deployment. *See* user-driven client deployments
semicolons 12
semi joins 51
SEQUENCE object 65
SEQUENCE objects 128
SERIALIZABLE isolation level 213, 218, 223–224, 229
Server Activity History report 353
SERVER DDL triggers 170–172
server principal 169, 171
servers
  linked 67
server-side tracing 286–289, 360
service tiers 335
session state management 244
SET command 134
SET SHOWPLAN_ALL ON statement 315
SET SHOWPLAN_TEXT ON statement 315
SET SHOWPLAN_XML ON statement 315
sharding 331–332
shared locks 229
simple indexes 25, 39
SIMPLE recovery model 338
slow queries 351
SMTP. *See* Single Mail Transfer Protocol (SMTP)
SNAPSHOT isolation level 219, 224–227, 229
snowflake schema 74
Sort operator 295–297

sorts 38–41
spatial data type 18
SPF. *See* send policy framework (SPF) records
Split-Merge service 333
sp_query_store_flush_db 305
sp_query_store_force_plan 305
sp_query_store_remove_plan 305
sp_query_store_remove_query 305
sp_query_store_reset_exec_stats 305
sp_query_store_unforce_plan 305
sp_trace_create 286
sp_trace_setevent 286
sp_trace_setfilter 286
sp_trace_setstatus 286
SQL Database. *See also* Azure SQL Database
  DMVs for 358
  Elastic Scale for 331–333
  Extended Events in 358
SQL Operating System (SQLOS) Scheduler 340
SQL Server
  baseline performance metrics 347–362
  Enterprise Edition 67
  managing database workload in 325–331
  memory optimization 339–340
  Standard Edition 67
SQL Server 2012
  columnstar indexes and 85
SQL Server 2014
  columnstar indexes and 85
SQL Server 2016
  columnstar indexes and 85–86
  editions 334–335
  service tiers 334–335
SQL Server Agent stored procedures 273
SQL Server Integration Services (SSIS) 275
SQL Server Lock Manager 212–213
SQL Server Management Studio 352

SQL Server Profiler 286, 288–289, 290
SQL Server Profiler deadlock graph 239–240
SQL Server Resource Governor 325–331
SQL Trace 283, 286–292, 347, 351, 355, 360
sql_variant 17
Standard edition 334
star schema 73–74
statistics 266–277
  accuracy of 266–273
  automatic updates 271–273
  data distribution and cardinality 267–271
  execution. *See* execution statistics
  maintenance tasks 273–276
  wait 340–343
statistics collection queries 258–259
STATS_DATE system function 272
storage
  troubleshooting 343–345
stored procedures 53, 101
  CLR 130
  complex business logic in 243
  creating 130–158
  designing, based on business requirements 131–135
  error handling in 144–158
  interpreted 249
  interpreted SQL 130
  natively compiled 130, 242, 247–252, 255–258
  parameters 132
    input and output 135–137
    table-valued 137–139
    type mismatch 143–144
  return codes 139–140
  returning data from 133–134
  server-side tracing 286
  SQL Server Agent extended 273
  streamlining logic 141–144
  structure of 131

transactions in 206–208
use of 131
string data 17
STRING_SPLIT() function 137–138
surrogate keys 6
surrogate keys 126–129
syntax
names 11–12
sys.database_connection_stats 358
sys.dm_db_index_physical_stats 279–280
sys.dm_db_index_usage_stats 276–278
sys.dm_db_missing_index_details 280
sys.dm_db_missing_index_groups 280
sys.dm_db_missing_index_group_stats 280
sys.dm_db_resource_stats 358
sys.dm_exec_query_stats 358
sys.dm_exec_session_wait_stats 342
sys.dm_io_virtual_file_stats 343
sys.dm_os_memory_cache_counters 345
sys.dm_os_memory_clerks 345
sys.dm_os_performance_counters 344
sys.dm_os_sys_memory 345
sys.dm_os_waiting_tasks 231, 234–235, 342
sys.dm_os_wait_stats 231, 235–236, 341–342
sys.dm_tran_locks 231, 231–234, 358
sys.event_log 358
sys.fn_trace_getinfo system function 288
sys.master_files 343
sys.query_store_plan 304
sys.query_store_query 304
sys.query_store_query_text 304
sys.query_store_runtime_stats 304
sys.query_store_runtime_stats_interval 305
sys.sp_xtp_control_proc_exec_stats 256
sys.sp_xtp_control_query_exec_stats 257–258
system functions
invalid use of, on search arguments 143

system health 359

T

table constraints 102
tables
  adding constraints to 119–122
  adding indexes to 32–40
  ALTER TABLE statement 14–15, 27
  cascading operations 113–115
  columns 5, 5–6
    adding 14–15
    data types for 15–24
    dropping 15
    indexed vs. included 41–45
  creating 11
  designing
    based on business requirements 2–4
    using normalization 4–11
  fact 74, 78
  hierarchies 115–116
  joins 35–37
  keys 5, 6
  memory-optimized 242–255
  modifying, using views 58–64
  multiple, in views 62–64
  names 11–13
  OLTP
    using non-clustered columnstore indexes on 81–85
  PRIMARY KEY constraints 27
  redundancy in 9
  relationship of non-key to key attributes 8–11
  rows 5–6
  rules covering shape of 5–7
  sorts 38–41
  temporary 244
  virtual 62
Table Scan operator 293

table-valued parameters 137–139, 244
table-valued user-defined functions 183–186
table variables 244
tempdb 224, 242, 247, 343
tempdb optimization 338–339
temporal extensions 14
temporary tables 244
third normal form 8, 9–11
THROW statement 145–147, 155, 157, 164
timestamp 18
time values 16–17
Top Resource Consuming Queries view 308
traces 283, 286–292, 351
  client-side 290–292, 360
  server-side 286–289, 360
Tracked Querie view 314
@@TRANCOUNT variable 204, 206, 207, 209
transaction control logic
  in stored procedures 144–158
transactions 195–216
  ACID properties of 196–203
  auto-commit 203
  distributed 208–209, 211, 219
  DLM statement results based on 196–203
  explicit 203–204, 205–209
  implicit 203–205, 216
  in high-concurrency databases 211–216
  isolation levels 216–229
  nested 206–207
  resource locks and 212–216
  savepoints within 209–211
Transact-SQL code 2–24, 12
Transact-SQL statements
  to add contraints to tables 119–122
triggers 101
  AFTER 176–179
  columnar data and 165–167

complex data integrity and 160–164
creating 159–179
DDL 159, 169–174
designing logic for, based on business requirements 159–169
DML 159–160, 169
INSTEAD OF 166–170, 176–179
login 159
logon 169, 174–176
running code in response to action 164–165
uses of 160
troubleshooting
IO 343–345
locking behavior 231–237
performance issues 352–355
storage 343–345
TRY...CATCH construct 149–151, 153–154, 157
T-SQL statements 346
tuple mover 72, 89

U

UDFs. *See* User-Defined Functions (UDFs)
uncommitable transactions 152
underscores 12
UNION ALL set operator 64
UNIQUE constraints 102, 120
FOREIGN KEY constraints relating to 117–119
use of 105–106
uniqueidentifier data type 18
uniqueness constraints
on indexes 27–28
updateable views 57–63
update locks 213
UPDATE operations
DEFAULT constraints on 104
in columnstore index 72
UPDATE statement 179
UPDATE STATISTICS statement 271, 275

Update Statistics Task dialog box 274
user accounts. *See also* identities
user-defined functions 50
User-Defined Functions (UDFs) 101, 159, 180–186
  classifier 328–329
  scalar 180–183
  table-valued 183–186
user-defined resource pools 327
user identities. *See* identities
user input
  DEFAULT constraints on 103–105
  limiting with constraints 107–111
user requirements
  creating views to meet 53–57

V

version store 227
VIEW_METADATA 52, 58
views 52–69
  basic form of 52
  designing
    based on user or business requirements 53–57
    updateable 57–63
  editable 58–64
  indexed 67–69
  layers of 55
  limiting what data can be added to table 61–62
  modifiable 167–169
  modifying data in, with multiple tables 62–64
  options for 52
  partitioned 64–67
  that reference single table, modifying 58–61
  uses 52, 53
    as reporting interface 55–56
    reformatting data in output 54–55
    table modification 58–64
    to hide data 53

virtual tables 62

W

wait statistics 340–343
wait types 235
Web edition 334
WHERE clause 62, 87, 88, 183
WIM. *See* Windows Imaging Format (WIM)
WITH CHECK OPTION clause 52, 61–62
WITH NATIVE_COMPILATION clause 249
WITH TIES in TOP clause 248
worker threads 340–341
workload groups 325, 328

X

XACT_ABORT() function 152, 155–156
XACT_STATE() function 152
XML data type 18
xml_deadlock_report 240

Y

ys.sp_xtp_control_query_exec_stats 256

About the authors

LOUIS DAVIDSON is a Microsoft MVP (Data Platform), and a Senior Data Architect for CBN. He has Over 20 years of experience with SQL Server as an architect, developer, and writer. Follow him on Twitter at @drsql.

STACIA VARGA, Microsoft MVP (Data Platform) is a consultant, educator, mentor, and author specializing in data solutions since 1999. She provides consulting and custom education services through her company, Data Inspirations, writes about her experiences with data at blog.datainspirations.com, and tweets as @_StaciaV_.

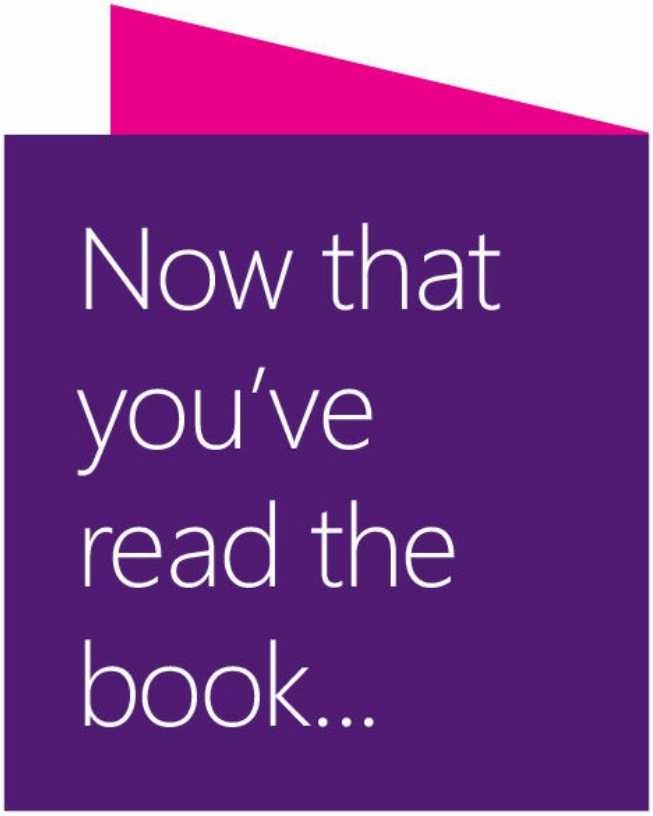
From technical overviews to drilldowns on special topics, get *free* ebooks from Microsoft Press at:

www.microsoftvirtualacademy.com/ebooks

Download your free ebooks in PDF, EPUB, and/or Mobi for Kindle formats.

Look for other great resources at Microsoft Virtual Academy, where you can learn new skills and help advance your career with free Microsoft training delivered by experts.

Microsoft Press



Tell us what you think!

Was it useful?
Did it teach you what you wanted to learn?
Was there room for improvement?

Let us know at https://aka.ms/tellpress

Your feedback goes directly to the staff at Microsoft Press,
and we read every one of your responses. Thanks in advance!

Code Snippets

Many titles include programming code or configuration examples. To optimize the presentation of these elements, view the eBook in single-column, landscape mode and adjust the font size to the smallest setting. In addition to presenting code and configurations in the reflowable text format, we have included images of the code that mimic the presentation found in the print book; therefore, where the reflowable format may compromise the presentation of the code listing, you will see a "Click here to view code image" link. Click the link to view the print-fidelity code image. To return to the previous page viewed, click the Back button on your device or app.

```
Name                                EmployeeNumber
---------------------------------- --------------
Harmetz, Adam                       000010012
Harmetz, Adam                       000010012
```

```
EmployeeId Name                           EmployeeNumber
---------- ------------------------------ ----------------------
         1 Harmetz, Adam                  000010012
         2 Harmetz, Adam                  000010012
         3 Popkova, Darya                 000000012
         4 Popkova, Darya                 000000013
```

```
EmployeeId Name                                EmployeeNumber
---------- ----------------------------------- ----------------------
         1 Harmetz, Adam                       000010012
         2 Popkova, Darya                      000000013
```

```
EmployeeId LastName      FirstName       EmployeeNumber
---------- ------------- --------------- --------------
         1 Harmetz       Adam            000010012
         2 Popkova       Darya           000000013
```

HumanResources.Employee (EmployeeNumber [key], LastName, FirstName)

Tag	AssociatedItemList
s344	mouse, keyboard

Tag	AssociatedItem1	AssociatedItem2	…	AssociatedItemN
s344	mouse	keyboard	…	not applicable

Tag	AssociatedItem1	AssociatedItem1Tag	AssociatedItem2	AssociatedItem2Tag
s344	mouse	r232	keyboard	q472

```
HumanResources.Employee (EmployeeNumber)

Equipment.Computer (Tag)
```

Tag (key attribute), ComputerType, TagCompany, TagCompanyURL, ComputerCost, PurchaseDate, AssignedEmployee

Equipment.Computer (Tag [key, ref to Tag], ComputerType, ComputerCost, PurchaseDate, AssignedEmployee [Reference to Employee]

Equipment.Tag (Tag [key], TagCompany, TagCompanyURL)

Equipment.Computer (Tag [key, Ref to Tag], ComputerType, ComputerCost, PurchaseDate, AssignedEmployee [Reference to Employee]

Equipment.TagCompany (TagCompany [key], TagCompanyURL)

Equipment.Tag (Tag [key], TagCompany [Reference to TagCompany])

Equipment.ComputerAssociatedItem (Tag [Reference to Computer], AssociatedItem, [key Tag, AssociatedItem)

HumanResources.Employee (EmployeeNumber [key], LastName, FirstName)

```
CREATE SCHEMA Examples;
GO --CREATE SCHEMA must be the only statement in the batch
```

```
CREATE TABLE Examples.Widget
(
    WidgetCode  varchar(10) NOT NULL
            CONSTRAINT PKWidget PRIMARY KEY,
    WidgetName  varchar(100) NULL
);
```

```
CREATE TABLE Examples.Widget
```

```
WidgetCode  varchar(10) NOT NULL
```

```
CONSTRAINT PKWidget PRIMARY KEY,
```

```
,CONSTRAINT PKWidget PRIMARY KEY (WidgetCode),
```

```
,CONSTRAINT PKWidget PRIMARY KEY (WidgetCode, WidgetName),
```

```
CREATE TABLE Examples.Widget
(
    WidgetCode  varchar(10) NOT NULL
            CONSTRAINT PKWidget PRIMARY KEY,
    WidgetName  varchar(100) NULL
) ON FileGroupName;
```

```
ALTER TABLE Examples.Widget
    ADD NullableColumn int NULL;
```

```
ALTER TABLE Examples.Widget
    ADD NotNullableColumn int NOT NULL
        CONSTRAINT DFLTWidget_NotNullableColumn DEFAULT ('Some Value');
```

To drop the column, you need to drop referencing constraints, which you also do with the ALTER TABLE statement:

```
ALTER TABLE Examples.Widget
    DROP DFLTWidget_NotNullableColumn;
```

```
ALTER TABLE Examples.Widget
    DROP COLUMN NotNullableColumn;
```

```
CREATE TABLE Examples.Company
(
        CompanyName     varchar(50) NOT NULL
                        CONSTRAINT PKCompany PRIMARY KEY
);
```

```
CREATE TABLE Examples.Payment
(
        PaymentNumber char(10) NOT NULL
                      CONSTRAINT PKPayment PRIMARY KEY,
        Amount int NOT NULL
);
```

```
CREATE TABLE Examples.ComputedColumn
(
     FirstName  nvarchar(50) NULL,
     LastName   nvarchar(50) NOT NULL,
     FullName AS CONCAT(LastName,',' + FirstName)
);
```

```
ALTER TABLE Examples.ComputedColumn DROP COLUMN FullName;

ALTER TABLE Examples.ComputedColumn
   ADD FullName AS CONCAT(LastName,', ' + FirstName) PERSISTED;
```

```
INSERT INTO Examples.ComputedColumn
VALUES (NULL,'Harris'),('Waleed','Heloo');
```

```
SELECT *
FROM   Examples.ComputedColumn;
```

```
FirstName    LastName       FullName
------------ -------------- --------------------
NULL         Harris         Harris
Waleed       Heloo          Heloo, Waleed
```

```
CREATE TABLE Examples.DataMasking
(
    FirstName    nvarchar(50) NULL,
    LastName    nvarchar(50) NOT NULL,
    PersonNumber char(10) NOT NULL,
    Status    varchar(10), --domain of values ('Active','Inactive','New')
    EmailAddress nvarchar(50) NULL, --(real email address ought to be longer)
    BirthDate date NOT NULL, --Time we first saw this person.
    CarCount   tinyint NOT NULL --just a count we can mask
);
INSERT INTO Examples.DataMasking(FirstName,LastName,PersonNumber, Status,
                                 EmailAddress, BirthDate, CarCount)
VALUES('Jay','Hamlin','0000000014','Active','jay@litwareinc.com','1979-01-12',0),
    ('Darya','Popkova','0000000032','Active','darya.p@proseware.net','1980-05-22', 1),
    ('Tomasz','Bochenek','0000000102','Active',NULL, '1959-03-30', 1);
```

```
CREATE USER MaskedView WITHOUT LOGIN;
GRANT SELECT ON Examples.DataMasking TO MaskedView;
```

```
ALTER TABLE Examples.DataMasking ALTER COLUMN FirstName
    ADD MASKED WITH (FUNCTION = 'default()');
ALTER TABLE Examples.DataMasking ALTER COLUMN BirthDate
    ADD MASKED WITH (FUNCTION = 'default()');
```

```
ALTER TABLE Examples.DataMasking ALTER COLUMN EmailAddress
    ADD MASKED WITH (FUNCTION = 'email()');
```

```
--Note that it uses double quotes in the function call
ALTER TABLE Examples.DataMasking ALTER COLUMN PersonNumber
    ADD MASKED WITH (FUNCTION = 'partial(2,"*******",1)');
```

```
ALTER TABLE Examples.DataMasking ALTER COLUMN LastName
    ADD MASKED WITH (FUNCTION = 'partial(3,"_____",2)');
```

```
ALTER TABLE Examples.DataMasking ALTER COLUMN Status
    ADD MASKED WITH (Function = 'partial(0,"Unknown",0)');
```

```
ALTER TABLE Examples.DataMasking ALTER COLUMN CarCount
    ADD MASKED WITH (FUNCTION = 'random(1,3)');
```

```
SELECT *
FROM   Examples.DataMasking;
```

```
FirstName LastName  PersonNumber Status     EmailAddress           BirthDate  CarCount
--------- --------- ------------ ---------- ---------------------- ---------- --------
Jay       Hamlin    0000000014   Active     jay@litwareinc.com     1979-01-12 0
Darya     Popkova   0000000032   Active     darya.p@proseware.net  1980-05-22 1
Tomasz    Bochenek  0000000102   Active     NULL                   1959-03-30 1
```

```
EXECUTE AS USER = 'MaskedView';
SELECT *
FROM   Examples.DataMasking;

FirstName LastName     PersonNumber Status  EmailAddress             BirthDate  CarCount
--------- ------------ ------------ ------- ------------------------ ---------- --------
xxxx      Hamlin       00****14     Unknown jXXX@XXXX.com            1900-01-01 2
xxxx      Popkova      00****32     Unknown dXXX@XXXX.com            1900-01-01 1
xxxx      Bochenek     00****02     Unknown NULL                     1900-01-01 1
```

```
CREATE TABLE Examples.UniquenessConstraint
(
    PrimaryUniqueValue int NOT NULL,
    AlternateUniqueValue1 int NULL,
    AlternateUniqueValue2 int NULL
);
```

```
ALTER TABLE Examples.UniquenessConstraint
    ADD CONSTRAINT PKUniquenessContraint PRIMARY KEY (PrimaryUniqueValue);
```

```
ALTER TABLE Examples.UniquenessConstraint
    ADD CONSTRAINT AKUniquenessContraint UNIQUE
          (AlternateUniqueValue1, AlternateUniqueValue2);
```

```
SELECT type_desc, is_primary_key, is_unique, is_unique_constraint
FROM   sys.indexes
WHERE  OBJECT_ID('Examples.UniquenessConstraint') = object_id;
```

```
name                    type_desc       is_primary_key is_unique is_unique_constraint
----------------------- --------------- -------------- --------- --------------------
PKUniquenessContraint   CLUSTERED        1              1         0
AKUniquenessContraint   NONCLUSTERED     0              1         1
```

```
INSERT INTO Examples.UniquenessConstraint
            (PrimaryUniqueValue, AlternateUniqueValue1, AlternateUniqueValue2)
VALUES (1, NULL, NULL), (2, NULL, NULL);
```

```
Msg 2627, Level 14, State 1, Line 95
Violation of UNIQUE KEY constraint 'AKUniquenessContraint'. Cannot insert duplicate key
in object 'Examples.UniquenessConstraint'. The duplicate key value is (<NULL>, <NULL>)
```

```
--Represents an order a person makes, there are 10,000,000 + rows in this table
CREATE TABLE Examples.Invoice
(
    InvoiceId   int NOT NULL CONSTRAINT PKInvoice PRIMARY KEY,
    --Other Columns Omitted
);
--Represents a type of discount the office gives a customer,
--there are 200 rows in this table
CREATE TABLE Examples.DiscountType
(
    DiscountTypeId   int NOT NULL CONSTRAINT PKDiscountType PRIMARY KEY,
    --Other Columns Omitted
)
--Represents the individual items that a customer has ordered, There is an average of
--3 items ordered per invoice, so there are over 30,000,000 rows in this table
CREATE TABLE Examples.InvoiceLineItem
(
   InvoiceLineItemId int NOT NULL CONSTRAINT PKInvoiceLineItem PRIMARY KEY,
   InvoiceId int NOT NULL
          CONSTRAINT FKInvoiceLineItem$Ref$Invoice
                REFERENCES Examples.Invoice (InvoiceId),
   DiscountTypeId int NOT NULL
          CONSTRAINT FKInvoiceLineItem$Ref$DiscountType
                REFERENCES Examples.DiscountType (DiscountTypeId)
    --Other Columns Omitted
);
```

```
CREATE INDEX InvoiceId ON Examples.InvoiceLineItem (InvoiceId);
```

```
CREATE INDEX DiscountTypeId ON Examples.InvoiceLineItem(DiscountTypeId)
                                          WHERE DiscountTypeId IS NOT NULL;
```

```
CREATE UNIQUE INDEX InvoiceColumns ON Examples.InvoiceLineItem(InvoiceId,
                                                               InvoiceLineItemId);
```

```
SELECT PaymentMethodId, COUNT(*) AS NumRows
FROM   Sales.CustomerTransactions
GROUP  BY PaymentMethodID;
```

PaymentMethodId	NumRows
4	26637
NULL	70510

```
SELECT *
FROM   Sales.CustomerTransactions
WHERE PaymentMethodID = 4;
```

```
SELECT CustomerID, OrderID, OrderDate, ExpectedDeliveryDate
FROM  Sales.Orders
WHERE CustomerPurchaseOrderNumber = '16374';
```

```
SET STATISTICS TIME ON;
SET STATISTICS IO ON;

SELECT CustomerID, OrderId, OrderDate, ExpectedDeliveryDate
FROM  Sales.Orders
WHERE CustomerPurchaseOrderNumber = '16374';

SET STATISTICS TIME OFF;
SET STATISTICS IO OFF;
```

```
Table 'Orders'. Scan count 1, logical reads 692, physical reads 0, read-ahead reads 0,
lob logical reads 0, lob physical reads 0, lob read-ahead reads 0.

 SQL Server Execution Times:
   CPU time = 15 ms,  elapsed time = 20 ms.:
```

```
CREATE INDEX CustomerPurchaseOrderNumber ON Sales.Orders(CustomerPurchaseOrderNumber);
```

```
Table 'Orders'. Scan count 1, logical reads 20, physical reads 0, read-ahead reads 0,
lob logical reads 0, lob physical reads 0, lob read-ahead reads 0.
 SQL Server Execution Times:
   CPU time = 0 ms,  elapsed time = 0 ms.
```

```
SELECT CONCAT(OBJECT_SCHEMA_NAME(object_id), '.', OBJECT_NAME(object_id)) AS TableName,
       name AS ColumnName, COLUMNPROPERTYEX(object_id, name, 'IsIndexable') AS Indexable
FROM   sys.columns
WHERE is_computed = 1;
```

```
DROP INDEX FK_Sales_Orders_ContactPersonID ON Sales.Orders;
```

```
SELECT OrderId, OrderDate, ExpectedDeliveryDate, People.FullName
FROM  Sales.Orders
        JOIN Application.People
            ON People.PersonID = Orders.ContactPersonID
WHERE  People.PreferredName = 'Aakriti';
```

```
Table 'Workfile'. Scan count 0, logical reads 0, physical reads 0, read-ahead reads 0,
lob logical reads 0, lob physical reads 0, lob read-ahead reads 0.

Table 'Worktable'. Scan count 0, logical reads 0, physical reads 0, read-ahead reads 0,
lob logical reads 0, lob physical reads 0,

lob read-ahead reads 0.

Table 'Orders'. Scan count 1, logical reads 692, physical reads 0, read-ahead reads 0,
lob logical reads 0, lob physical reads 0,

lob read-ahead reads 0.

Table 'People'. Scan count 1, logical reads 80, physical reads 0, read-ahead reads 0,
lob logical reads 0, lob physical reads 0,

lob read-ahead reads 0.

CPU time = 15 ms,  elapsed time = 53 ms.
```

```
CREATE INDEX FK_Sales_Orders_ContactPersonID ON Sales.Orders
--Note that USERDATA is a filegroup where the index was originally
              (ContactPersonID ASC ) ON USERDATA;
```

```
Table 'Orders'. Scan count 2, logical reads 695, physical reads 0, read-ahead reads 0,
lob logical reads 0, lob physical reads 0,

lob read-ahead reads 0.

Table 'People'. Scan count 1, logical reads 80, physical reads 0, read-ahead reads 0,
lob logical reads 0, lob physical reads 0,

lob read-ahead reads 0.

CPU time = 0 ms,  elapsed time = 17 ms.
```

```
CREATE INDEX PreferredName ON Application.People (PreferredName) ON USERDATA;
```

```
Table 'Orders'. Scan count 2, logical reads 695, physical reads 0, read-ahead reads 0,
lob logical reads 0, lob physical reads 0,

lob read-ahead reads 0.

Table 'People'. Scan count 1, logical reads 6, physical reads 0, read-ahead reads 0, lob
logical reads 0, lob physical reads 0, lob read-ahead reads 0.

CPU time = 0 ms,  elapsed time = 19 ms.
```

```
SELECT SalespersonPersonId, OrderDate
FROM Sales.Orders
ORDER BY SalespersonPersonId ASC, OrderDate ASC;
```

```
Table 'Worktable'. Scan count 0, logical reads 0, physical reads 0, read-ahead reads 0,
lob logical reads 0, lob physical reads 0,

lob read-ahead reads 0.

Table 'Orders'. Scan count 3, logical reads 758, physical reads 0, read-ahead reads 0,
lob logical reads 0, lob physical reads 0,

lob read-ahead reads 0.

CPU time = 94 ms,  elapsed time = 367 ms.
```

```
CREATE INDEX SalespersonPersonID_OrderDate ON Sales.Orders
                                (SalespersonPersonID ASC, OrderDate ASC);
```

```
Table 'Orders'. Scan count 1, logical reads 157, physical reads 0, read-ahead reads 0,
lob logical reads 0, lob physical reads 0, lob read-ahead reads 0.

CPU time = 47 ms,  elapsed time = 217 ms.
```

```
SELECT SalespersonPersonId, OrderDate
FROM Sales.Orders
ORDER BY SalespersonPersonId DESC, OrderDate DESC;
```

```
SELECT Orders.ContactPersonID, People.PersonID
FROM   Sales.Orders
         INNER JOIN Application.People
            ON Orders.ContactPersonID = People.PersonID;
```

```
SELECT OrderId, OrderDate, ExpectedDeliveryDate, People.FullName
FROM  Sales.Orders
        JOIN Application.People
            ON People.PersonID = Orders.ContactPersonID
WHERE  People.PreferredName = 'Aakriti';
```

```
Table 'Orders'. Scan count 2, logical reads 695, physical reads 0, read-ahead reads 0,
lob logical reads 0, lob physical reads 0,

lob read-ahead reads 0.

Table 'People'. Scan count 1, logical reads 6, physical reads 0, read-ahead reads 0, lob
logical reads 0, lob physical reads 0,

lob read-ahead reads 0.

CPU time = 0 ms,  elapsed time = 141 ms.
```

```
SELECT Orders.ContactPersonId, People.PreferredName
FROM  Sales.Orders
        JOIN Application.People
            ON People.PersonID = Orders.ContactPersonID
WHERE  People.PreferredName = 'Aakriti';
```

```
Table 'Orders'. Scan count 2, logical reads 4, physical reads 0, read-ahead reads 0, lob
logical reads 0, lob physical reads 0,

lob read-ahead reads 0.

Table 'People'. Scan count 1, logical reads 2, physical reads 0, read-ahead reads 0, lob
logical reads 0, lob physical reads 0,

lob read-ahead reads 0.

CPU time = 0 ms,  elapsed time = 38 ms.
```

```
CREATE NONCLUSTERED INDEX ContactPersonID_Include_OrderDate_ExpectedDeliveryDate
ON Sales.Orders ( ContactPersonID )
INCLUDE ( OrderDate,ExpectedDeliveryDate)
ON USERDATA;
GO
```

```
DROP INDEX PreferredName ON Application.People;
GO
CREATE NONCLUSTERED INDEX PreferredName_Include_FullName
ON Application.People (   PreferredName )
INCLUDE (FullName)
ON USERDATA;
```

```
SELECT OrderId, OrderDate, ExpectedDeliveryDate, People.FullName
FROM  Sales.Orders
        JOIN Application.People
            ON People.PersonID = Orders.ContactPersonID
WHERE  People.PreferredName = 'Aakriti';
```

```
Table 'Orders'. Scan count 2, logical reads 6, physical reads 0, read-ahead reads 0, lob
logical reads 0, lob physical reads 0,

lob read-ahead reads 0.

Table 'People'. Scan count 1, logical reads 2, physical reads 0, read-ahead reads 0, lob
logical reads 0, lob physical reads 0,

lob read-ahead reads 0.

CPU time = 0 ms,  elapsed time = 79 ms.
```

```
FROM   Sales.CustomerTransactions
WHERE PaymentMethodID = 4;
```

```
CREATE NONCLUSTERED INDEX [<Name of Missing Index, sysname,>]
ON [Sales].[CustomerTransactions] ([PaymentMethodID])
INCLUDE ([CustomerTransactionID],[CustomerID],[TransactionTypeID], [InvoiceID],[Transact
ionDate],[AmountExcludingTax],[TaxAmount],[TransactionAmount],
[OutstandingBalance],[FinalizationDate],[IsFinalized],[LastEditedBy],[LastEditedWhen])
```

```
SELECT OrderDate, ExpectedDeliveryDate
FROM  Sales.Orders
WHERE OrderDate > '2015-01-01';
```

```
CREATE NONCLUSTERED INDEX [<Name of Missing Index, sysname,>]
ON [Sales].[Orders] ([OrderDate])
INCLUDE ([ExpectedDeliveryDate]);
```

```
--2074 Rows
SELECT *
INTO   Examples.PurchaseOrders
FROM   WideWorldImporters.Purchasing.PurchaseOrders;

ALTER  TABLE Examples.PurchaseOrders
    ADD CONSTRAINT PKPurchaseOrders PRIMARY KEY (PurchaseOrderId);

--8367 Rows
SELECT *
INTO   Examples.PurchaseOrderLines
FROM   WideWorldImporters.Purchasing.PurchaseOrderLines;

ALTER  TABLE Examples.PurchaseOrderLines
    ADD CONSTRAINT PKPurchaseOrderLines PRIMARY KEY (PurchaseOrderLineID);

ALTER TABLE Examples.PurchaseOrderLines
    ADD CONSTRAINT FKPurchaseOrderLines_Ref_Examples_PurchaseOrderLines
        FOREIGN KEY (PurchaseOrderId) REFERENCES
                                    Examples.PurchaseOrders(PurchaseOrderId);
```

```
SELECT *
FROM   Examples.PurchaseOrders
WHERE  PurchaseOrders.OrderDate BETWEEN '2016-03-10' AND '2016-03-14';

SELECT PurchaseOrderId, ExpectedDeliveryDate
FROM   Examples.PurchaseOrders
WHERE  EXISTS (SELECT *
               FROM  Examples.PurchaseOrderLines
               WHERE PurchaseOrderLines.PurchaseOrderId =
                                          PurchaseOrders.PurchaseOrderID)
  AND  PurchaseOrders.OrderDate BETWEEN '2016-03-10' AND '2016-03-14' ;
```

```
CREATE INDEX PurchaseOrderId ON Examples.PurchaseOrderLines (PurchaseOrderId);
```

```
CREATE INDEX OrderDate ON Examples.PurchaseOrders (OrderDate);
```

```
CREATE INDEX OrderDate_Incl_ExpectedDeliveryDate
     ON Examples.PurchaseOrders (OrderDate) INCLUDE (ExpectedDeliveryDate);
```

```
CREATE VIEW SchemaName.ViewName
[WITH OPTIONS]
AS SELECT statement
[WITH CHECK OPTION]
```

```
CREATE VIEW Sales.Orders12MonthsMultipleItems
AS
SELECT OrderId, CustomerID, SalespersonPersonID, OrderDate, ExpectedDeliveryDate
FROM   Sales.Orders
WHERE  OrderDate >= DATEADD(Month,-12,SYSDATETIME())
  AND (SELECT COUNT(*)
       FROM   Sales.OrderLines
       WHERE  OrderLines.OrderID = Orders.OrderID) > 1;
```

```
SELECT TOP 5 *
FROM   Sales.Orders12MonthsMultipleItems
ORDER BY ExpectedDeliveryDate desc;
```

OrderId	CustomerID	SalespersonPersonID	OrderDate	ExpectedDeliveryDate
73550	967	15	2016-05-31	2016-06-01
73549	856	16	2016-05-31	2016-06-01
73548	840	3	2016-05-31	2016-06-01
73547	6	14	2016-05-31	2016-06-01
73546	810	3	2016-05-31	2016-06-01

```
SELECT PersonId, IsPermittedToLogon, IsEmployee, IsSalesPerson
FROM   Application.People;
```

```
PersonId    IsPermittedToLogon IsEmployee IsSalesPerson
----------- ------------------ ---------- -------------
1           0                  0          0
2           1                  1          1
3           1                  1          1
4           1                  1          0
```

```
CREATE VIEW Application.PeopleEmployeeStatus
AS
SELECT PersonId, FullName,
       IsPermittedToLogon, IsEmployee, IsSalesPerson,
       CASE WHEN IsPermittedToLogon = 1 THEN 'Can Logon'
             ELSE 'Can''t Logon' END AS LogonRights,
       CASE WHEN IsEmployee = 1 and IsSalesPerson = 1
                THEN 'Sales Person'
            WHEN IsEmployee = 1
                THEN 'Regular'
            ELSE 'Not Employee' END AS EmployeeType
FROM   Application.People;
```

```
SELECT PersonId, LogonRights, EmployeeType
FROM   Application.PeopleEmployeeStatus;
```

```
PersonId    LogonRights EmployeeType
----------- ----------- -------------
1           Can't Logon Not Employee
2           Can Logon   Sales Person
3           Can Logon   Sales Person
4           Can Logon   Regular
```

```
CREATE SCHEMA Reports;
GO
CREATE VIEW Reports.InvoiceSummaryBasis
AS
SELECT Invoices.InvoiceId, CustomerCategories.CustomerCategoryName,
       Cities.CityName, StateProvinces.StateProvinceName,
       StateProvinces.SalesTerritory,
       Invoices.InvoiceDate,
       --the grain of the report is at the invoice, so total
       --the amounts for invoice
       SUM(InvoiceLines.LineProfit) as InvoiceProfit,
       SUM(InvoiceLines.ExtendedPrice) as InvoiceExtendedPrice
FROM   Sales.Invoices
         JOIN Sales.InvoiceLines
            ON Invoices.InvoiceID = InvoiceLines.InvoiceID
         JOIN Sales.Customers
              ON Customers.CustomerID = Invoices.CustomerID
         JOIN Sales.CustomerCategories
              ON Customers.CustomerCategoryID =
                                 CustomerCategories.CustomerCategoryID
         JOIN Application.Cities
              ON Customers.DeliveryCityID = Cities.CityID
         JOIN Application.StateProvinces
              ON StateProvinces.StateProvinceID = Cities.StateProvinceID
GROUP BY Invoices.InvoiceId, CustomerCategories.CustomerCategoryName,
       Cities.CityName, StateProvinces.StateProvinceName,
       StateProvinces.SalesTerritory,
       Invoices.InvoiceDate;
```

```
SELECT TOP 5 SalesTerritory, SUM(InvoiceProfit) AS InvoiceProfitTotal
FROM Reports.InvoiceSummaryBasis
WHERE InvoiceDate > '2016-05-01'
GROUP BY SalesTerritory
ORDER BY InvoiceProfitTotal DESC;
```

SalesTerritory	InvoiceProfitTotal
Southeast	536367.60
Great Lakes	366182.65
Mideast	344703.00
Southwest	344386.95
Plains	288766.20

```
SELECT TOP 5 StateProvinceName, CustomerCategoryName,
       SUM(InvoiceExtendedPrice) AS InvoiceExtendedPriceTotal
FROM Reports.InvoiceSummaryBasis
WHERE InvoiceDate > '2016-05-01'
GROUP BY StateProvinceName, CustomerCategoryName
ORDER BY InvoiceExtendedPriceTotal DESC;
```

```
StateProvinceName    CustomerCategoryName   InvoiceExtendedPriceTotal
-------------------- ---------------------- ----------------------------------------
Texas                Novelty Shop           229966.31
Pennsylvania         Novelty Shop           210254.62
Ohio                 Novelty Shop           201242.59
New York             Novelty Shop           197664.32
California           Novelty Shop           178698.48
```

```
CREATE NONCLUSTERED INDEX [<Name of Missing Index, sysname,>]
        ON [Sales].[Invoices] ([InvoiceDate]) INCLUDE ([InvoiceID],[CustomerID]);
```

```
CREATE TABLE Examples.Gadget
(
    GadgetId    int NOT NULL CONSTRAINT PKGadget PRIMARY KEY,
    GadgetNumber char(8) NOT NULL CONSTRAINT AKGadget UNIQUE,
    GadgetType  varchar(10) NOT NULL
);
INSERT INTO Examples.Gadget(GadgetId, GadgetNumber, GadgetType)
VALUES  (1,'00000001','Electronic'),
        (2,'00000002','Manual'),
        (3,'00000003','Manual');
```

```
CREATE VIEW Examples.ElectronicGadget
AS
    SELECT GadgetId, GadgetNumber, GadgetType,
           UPPER(GadgetType) AS UpperGadgedType
    FROM   Examples.Gadget
    WHERE GadgetType = 'Electronic';
```

```
SELECT ElectronicGadget.GadgetNumber AS FromView, Gadget.GadgetNumber AS FromTable,
        Gadget.GadgetType, ElectronicGadget.UpperGadgetType
FROM   Examples.ElectronicGadget
         FULL OUTER JOIN Examples.Gadget
            ON ElectronicGadget.GadgetId = Gadget.GadgetId;
```

```
FromView FromTable GadgetType UpperGadgetType
-------- --------- ---------- --------------
00000001 00000001  Electronic ELECTRONIC
NULL     00000002  Manual     NULL
NULL     00000003  Manual     NULL
```

```
INSERT INTO Examples.ElectronicGadget(GadgetId, GadgetNumber,
                                      GadgetType, UpperGadgetType)
VALUES (4,'00000004','Electronic','XXXXXXXXXX'), --row we can see in view
       (5,'00000005','Manual','YYYYYYYYYY'); --row we cannot see in view
```

```
Msg 4406, Level 16, State 1, Line 433
Update or insert of view or function 'Examples.ElectronicGadget' failed because
it contains a derived or constant field.
```

```
INSERT INTO Examples.ElectronicGadget(GadgetId, GadgetNumber, GadgetType)
VALUES (4,'00000004','Electronic'),
       (5,'00000005','Manual');
```

```
SELECT ElectronicGadget.GadgetNumber as FromView, Gadget.GadgetNumber as FromTable,
        Gadget.GadgetType, ElectronicGadget.UpperGadgetType
FROM   Examples.ElectronicGadget
         FULL OUTER JOIN Examples.Gadget
            ON ElectronicGadget.GadgetId = Gadget.GadgetId;

WHERE Gadget.GadgetId in (4,5);
```

Both rows were created, even though you cannot see the row in the view after the operation:

```
FromView FromTable GadgetType UpperGadgetType
-------- --------- ---------- ---------------
00000004 00000004  Electronic ELECTRONIC
NULL     00000005  Manual     NULL
```

```
--Update the row we could see to values that could not be seen
UPDATE Examples.ElectronicGadget
SET    GadgetType   = 'Manual'
WHERE  GadgetNumber = '00000004';

--Update the row we could see to values that could actually see
UPDATE Examples.ElectronicGadget
SET    GadgetType   = 'Electronic'
WHERE  GadgetNumber = '00000005';
```

```
FromView FromTable GadgetType UpperGadgetType
-------- --------- ---------- ---------------
NULL     00000004  Manual     NULL
NULL     00000005  Manual     NULL
```

```
UPDATE Examples.Gadget
SET    GadgetType   = 'Electronic'
WHERE  GadgetNumber = '00000004';
```

```
ALTER VIEW Examples.ElectronicGadget
AS
    SELECT GadgetId, GadgetNumber, GadgetType,
           UPPER(GadgetType) AS UpperGadgetType
    FROM   Examples.Gadget
    WHERE GadgetType = 'Electronic'
    WITH CHECK OPTION;
```

```
INSERT INTO Examples.ElectronicGadget(GadgetId, GadgetNumber, GadgetType)
VALUES (6,'00000006','Manual');
```

```
Msg 550, Level 16, State 1, Line 482
The attempted insert or update failed because the target view either specifies
WITH CHECK OPTION or spans a view that specifies WITH CHECK OPTION and one or more
rows resulting from the operation did not qualify under the CHECK OPTION constraint.
```

```
UPDATE Examples.ElectronicGadget
SET    GadgetType   = 'Manual'
WHERE  GadgetNumber = '00000004';
```

```
CREATE TABLE Examples.GadgetType
(
    GadgetType  varchar(10) NOT NULL CONSTRAINT PKGadgetType PRIMARY KEY,
    Description varchar(200) NOT NULL
)
INSERT INTO Examples.GadgetType(GadgetType, Description)
VALUES ('Manual','No batteries'),
       ('Electronic','Lots of bats');

ALTER TABLE Examples.Gadget
   ADD CONSTRAINT FKGadget$ref$Examples_GadgetType
       FOREIGN KEY (GadgetType) REFERENCES Examples.GadgetType (GadgetType);
```

```
CREATE VIEW Examples.GadgetExtension
AS
    SELECT Gadget.GadgetId, Gadget.GadgetNumber,
           Gadget.GadgetType, GadgetType.GadgetType As DomainGadgetType,
           GadgetType.Description as GadgetTypeDescription
    FROM   Examples.Gadget
             JOIN Examples.GadgetType
                ON Gadget.GadgetType = GadgetType.GadgetType;
```

```
INSERT INTO Examples.GadgetExtension(GadgetId, GadgetNumber, GadgetType,
                    DomainGadgetType, GadgetTypeDescription)
VALUES(7,'00000007','Acoustic','Acoustic','Sound');
```

```
Msg 4405, Level 16, State 1, Line 512
View or function 'Examples.GadgetExtension' is not updatable because the modification
affects multiple base tables.
```

```
INSERT INTO Examples.GadgetExtension(DomainGadgetType, GadgetTypeDescription)
VALUES('Acoustic','Sound');

INSERT INTO Examples.GadgetExtension(GadgetId, GadgetNumber, GadgetType)
VALUES(7,'00000007','Acoustic');
```

```
SELECT *
FROM   Examples.Gadget
             JOIN Examples.GadgetType
                ON Gadget.GadgetType = GadgetType.GadgetType
WHERE  Gadget.GadgetType = 'Electronic';
```

GadgetId	GadgetNumber	GadgetType	GadgetType	Description
1	00000001	Electronic	Electronic	Lots of bats
4	00000004	Electronic	Electronic	Lots of bats

```
UPDATE Examples.GadgetExtension
SET   GadgetTypeDescription = 'Uses Batteries'
WHERE GadgetId = 1;
```

```
GadgetId    GadgetNumber GadgetType GadgetType Description
----------- ------------ ---------- ---------- ---------------
1           00000001     Electronic Electronic Uses Batteries
4           00000004     Electronic Electronic Uses Batteries
```

```
CREATE TABLE Examples.Invoices_Region1
(
    InvoiceId   int NOT NULL
        CONSTRAINT PKInvoices_Region1 PRIMARY KEY,
        CONSTRAINT CHKInvoices_Region1_PartKey
                        CHECK (InvoiceId BETWEEN 1 and 10000),
    CustomerId  int NOT NULL,
    InvoiceDate date NOT NULL
);
CREATE TABLE Examples.Invoices_Region2
(
    InvoiceId   int NOT NULL
        CONSTRAINT PKInvoices_Region2 PRIMARY KEY,
        CONSTRAINT CHKInvoices_Region2_PartKey
                        CHECK (InvoiceId BETWEEN 10001 and 20000),

          CustomerId  int NOT NULL,
          InvoiceDate date NOT NULL

      );

      INSERT INTO Examples.Invoices_Region1 (InvoiceId, CustomerId, InvoiceDate)
      SELECT InvoiceId, CustomerId, InvoiceDate
      FROM   WideWorldImporters.Sales.Invoices
      WHERE  InvoiceId BETWEEN 1 and 10000;

      INSERT INTO Examples.Invoices_Region2 (InvoiceId, CustomerId, InvoiceDate)
      SELECT InvoiceId, CustomerId, InvoiceDate
      FROM   WideWorldImporters.Sales.Invoices
      WHERE  InvoiceId BETWEEN 10001 and 20000;
```

```
CREATE VIEW Examples.InvoicesPartitioned
AS
    SELECT InvoiceId, CustomerId, InvoiceDate
    FROM   Examples.Invoices_Region1
    UNION ALL
    SELECT InvoiceId, CustomerId, InvoiceDate
    FROM   Examples.Invoices_Region2;
```

```
SELECT *
FROM  Examples.InvoicesPartitioned
WHERE InvoiceId = 1;
```

```
SELECT InvoiceId
FROM   Examples.InvoicesPartitioned
WHERE  InvoiceDate = '2013-01-01';
```

```
SELECT InvoiceId, CustomerId, InvoiceDate
FROM   Sales.Invoices_Region1
UNION ALL
SELECT InvoiceId, CustomerId, InvoiceDate
FROM   ServerName.DatabaseName.Sales.Invoices_Region2;
```

```
CREATE VIEW Sales.InvoiceCustomerInvoiceAggregates
WITH SCHEMABINDING
AS
SELECT Invoices.CustomerId,
       SUM(ExtendedPrice * Quantity) AS SumCost,
       SUM(LineProfit) AS SumProfit,
       COUNT_BIG(*) AS TotalItemCount
FROM  Sales.Invoices
          JOIN Sales.InvoiceLines
                ON  Invoices.InvoiceID = InvoiceLines.InvoiceID
GROUP  BY Invoices.CustomerID;
```

```
SELECT *
FROM   Sales.InvoiceCustomerInvoiceAggregates;
```

```
CREATE UNIQUE CLUSTERED INDEX XPKInvoiceCustomerInvoiceAggregates on
                        Sales.InvoiceCustomerInvoiceAggregates(CustomerID);
```

```
SELECT Invoices.CustomerId,
       SUM(ExtendedPrice * Quantity) / SUM(LineProfit),
       COUNT(*) AS TotalItemCount
FROM  Sales.Invoices
          JOIN Sales.InvoiceLines
                ON  Invoices.InvoiceID = InvoiceLines.InvoiceID
GROUP  BY Invoices.CustomerID;
```

```
CREATE TABLE Fact.[Order]
(
    [Order Key] bigint IDENTITY(1,1) NOT NULL,
    [City Key] int NOT NULL,
    [Customer Key] int NOT NULL,
    [Stock Item Key] int NOT NULL,
    [Order Date Key] date NOT NULL,
    [Picked Date Key] date NULL,
    [Salesperson Key] int NOT NULL,
    [Picker Key] int NULL,
    [WWI Order ID] int NOT NULL,
    [WWI Backorder ID] int NULL,
    [Description] nvarchar(100) NOT NULL,
    [Package] nvarchar(50) NOT NULL,
    [Quantity] int NOT NULL,
    [Unit Price] decimal(18, 2) NOT NULL,
    [Tax Rate] decimal(18, 3) NOT NULL,
    [Total Excluding Tax] decimal(18, 2) NOT NULL,
    [Tax Amount] decimal(18, 2) NOT NULL,
    [Total Including Tax] decimal(18, 2) NOT NULL,
    [Lineage Key] int NOT NULL
);
```

```
CREATE TABLE Dimension.Customer
(
    [Customer Key] int NOT NULL,
    [WWI Customer ID] int NOT NULL,
    [Customer] nvarchar(100) NOT NULL,
    [Bill To Customer] nvarchar(100) NOT NULL,
    [Category] nvarchar(50) NOT NULL,
    [Buying Group] nvarchar(50) NOT NULL,
    [Primary Contact] nvarchar(50) NOT NULL,
    [Postal Code] nvarchar(10) NOT NULL,
    [Valid From] datetime2(7) NOT NULL,
    [Valid To] datetime2(7) NOT NULL,
    [Lineage Key] int NOT NULL
);
CREATE TABLE Dimension.Date(
    Date date NOT NULL,
    [Day Number] int NOT NULL,
    [Day] nvarchar(10) NOT NULL,
    [Month] nvarchar(10) NOT NULL,
    [Short Month] nvarchar(3) NOT NULL,
    [Calendar Month Number] int NOT NULL,
    [Calendar Month Label] nvarchar(20) NOT NULL,
    [Calendar Year] int NOT NULL,
    [Calendar Year Label] nvarchar(10) NOT NULL,
    [Fiscal Month Number] int NOT NULL,
    [Fiscal Month Label] nvarchar(20) NOT NULL,
    [Fiscal Year] int NOT NULL,
    [Fiscal Year Label] nvarchar(10) NOT NULL,
    [ISO Week Number] int NOT NULL
);
```

```
DROP INDEX [CCX_Fact_Order] ON [Fact].[Order];
```

```
SELECT Customer.Category, Date.[Calendar Month Number],
        COUNT(*) AS SalesCount,
        SUM([Total Excluding Tax]) as SalesTotal
FROM   Fact.[Order]
         JOIN Dimension.Date
            ON Date.Date = [Order].[Order Date Key]
         JOIN Dimension.Customer
            ON Customer.[Customer Key] = [Order].[Customer Key]
GROUP BY Customer.Category, Date.[Calendar Month Number]
ORDER BY Category, Date.[Calendar Month Number], SalesCount, SalesTotal;
```

```
Table 'Customer'. Scan count 3, logical reads 40, physical reads 0, read-ahead reads 0,
lob logical reads 0, lob physical reads 0,

lob read-ahead reads 0.

Table 'Date'. Scan count 3, logical reads 79, physical reads 0, read-ahead reads 0, lob
logical reads 0, lob physical reads 0,

lob read-ahead reads 0.

Table 'Order'. Scan count 7, logical reads 5908, physical reads 0, read-ahead reads 0,
lob logical reads 0, lob physical reads 0,

lob read-ahead reads 0.

Table 'Workfile'. Scan count 0, logical reads 0, physical reads 0, read-ahead reads 0,
lob logical reads 0, lob physical reads 0,

lob read-ahead reads 0.

Table 'Worktable'. Scan count 0, logical reads 0, physical reads 0, read-ahead reads 0,
lob logical reads 0, lob physical reads 0,

lob read-ahead reads 0.

CPU time = 344 ms,  elapsed time = 276 ms.
```

```
CREATE NONCLUSTERED INDEX SpecificQuery ON [Fact].[Order] ([Customer Key])
INCLUDE ([Order Date Key],[Total Excluding Tax]);
```

```
CREATE CLUSTERED COLUMNSTORE INDEX [CCX_Fact_Order] ON [Fact].[Order];
```

```
Table 'Order'. Scan count 1, logical reads 0, physical reads 0, read-ahead reads 0, lob
logical reads 256, lob physical reads 0,

lob read-ahead reads 0.

Table 'Order'. Segment reads 4, segment skipped 0.

Table 'Worktable'. Scan count 0, logical reads 0, physical reads 0, read-ahead reads 0,
lob logical reads 0, lob physical reads 0,

lob read-ahead reads 0.

Table 'Customer'. Scan count 1, logical reads 15, physical reads 0, read-ahead reads 0,
lob logical reads 0, lob physical reads 0,

lob read-ahead reads 0.

Table 'Date'. Scan count 1, logical reads 28, physical reads 0, read-ahead reads 0, lob
logical reads 0, lob physical reads 0,

lob read-ahead reads 0.

CPU time = 15 ms,  elapsed time = 65 ms.
```

```
CREATE NONCLUSTERED COLUMNSTORE INDEX [NCCX_Fact_Order] ON [Fact].[Order] (
       [Order Key] ,[City Key] ,[Customer Key] ,[Stock Item Key]
      ,[Order Date Key] ,[Picked Date Key] ,[Salesperson Key] ,[Picker Key]
      ,[WWI Order ID],[WWI Backorder ID],[Package]
      ,[Quantity],[Unit Price],[Tax Rate],[Total Excluding Tax]
      ,[Tax Amount],[Total Including Tax]);
```

```
SELECT Customer.Category, Date.[Calendar Year],
        Date.[Calendar Month Number],
        COUNT(*) as SalesCount,
        SUM([Total Excluding Tax]) AS SalesTotal,
        AVG([Total Including Tax]) AS AvgWithTaxTotal,
        MAX(Date.Date) AS MaxOrderDate
FROM   Fact.[Order]
         JOIN Dimension.Date
            ON Date.Date = [Order].[Order Date Key]
         JOIN Dimension.Customer
            ON Customer.[Customer Key] = [Order].[Customer Key]
GROUP BY Customer.Category, Date.[Calendar Year], Date.[Calendar Month Number]
ORDER BY Category, Date.[Calendar Month Number], SalesCount, SalesTotal;
```

```
CREATE TABLE Sales.InvoiceLines
(
    InvoiceLineID int NOT NULL,
    InvoiceID int NOT NULL,
    StockItemID int NOT NULL,
    Description nvarchar(100) NOT NULL,
    PackageTypeID int NOT NULL,
    Quantity int NOT NULL,
    UnitPrice decimal(18, 2) NULL,
    TaxRate decimal(18, 3) NOT NULL,
    TaxAmount decimal(18, 2) NOT NULL,
    LineProfit decimal(18, 2) NOT NULL,
    ExtendedPrice decimal(18, 2) NOT NULL,
    LastEditedBy int NOT NULL,
    LastEditedWhen datetime2(7) NOT NULL,
 CONSTRAINT PK_Sales_InvoiceLines PRIMARY KEY
              CLUSTERED ( InvoiceLineID )
 );
--Not shown: FOREIGN KEY constraints, indexes other than the PK

CREATE NONCLUSTERED COLUMNSTORE INDEX NCCX_Sales_OrderLines ON Sales.OrderLines
(
    OrderID,
    StockItemID,
    Description,
    Quantity,
    UnitPrice,
    PickedQuantity
) ON USERDATA;
```

```
CREATE NONCLUSTERED COLUMNSTORE INDEX NCCX_Sales_OrderLines ON Sales.OrderLines
(
        OrderID,
        StockItemID,
        Description,
        Quantity,
        UnitPrice,
        PickedQuantity
) WITH (DROP_EXISTING = ON, COMPRESSION_DELAY = 5) ON USERDATA;
```

```
CREATE TABLE Sales.Orders
(
    OrderID int NOT NULL,
    CustomerID int NOT NULL,
    SalespersonPersonID int NOT NULL,
    PickedByPersonID int NULL,
    ContactPersonID int NOT NULL,
    BackorderOrderID int NULL,
    OrderDate date NOT NULL,
    ExpectedDeliveryDate date NOT NULL,
    CustomerPurchaseOrderNumber nvarchar(20) NULL,
    IsUndersupplyBackordered bit NOT NULL,
    Comments nvarchar(max) NULL,
    DeliveryInstructions nvarchar(max) NULL,
    InternalComments nvarchar(max) NULL,
    PickingCompletedWhen datetime2(7) NULL,
    LastEditedBy int NOT NULL,
    LastEditedWhen datetime2(7) NOT NULL,
    CONSTRAINT PK_Sales_Orders PRIMARY KEY CLUSTERED
    (
       OrderID ASC
    )
);
```

```
CREATE NONCLUSTERED COLUMNSTORE INDEX NCCI_Orders ON Sales.Orders
(
	PickedByPersonId,
	SalespersonPersonID,
	OrderDate,
	PickingCompletedWhen
)
WHERE PickedByPersonId IS NOT NULL;
```

```
SELECT *
INTO   Fact.SaleBase
FROM   Fact.Sale;

CREATE CLUSTERED COLUMNSTORE INDEX CColumnsStore ON Fact.SaleBase;
```

```
SELECT Date.[Fiscal Year], Customer.Category,  Sum(Quantity) as NumSales
FROM   Fact.SaleBase
         JOIN Dimension.Customer
            on Customer.[Customer Key] = SaleBase.[Customer Key]
         JOIN Dimension.Date
            ON Date.Date = SaleBase.[Invoice Date Key]
GROUP BY Date.[Fiscal Year], Customer.Category
ORDER BY Date.[Fiscal Year], Customer.Category;
```

```
Table 'Worktable'. Scan count 0, logical reads 0, physical reads 0, read-ahead reads 0,
lob logical reads 0, lob physical reads 0,

lob read-ahead reads 0.

Table 'Customer'. Scan count 1, logical reads 15, physical reads 0, read-ahead reads 0,
lob logical reads 0, lob physical reads 0,

lob read-ahead reads 0.

Table 'Date'. Scan count 1, logical reads 28, physical reads 0, read-ahead reads 0, lob
logical reads 0, lob physical reads 0,

lob read-ahead reads 0.

CPU time = 15 ms,  elapsed time = 22 ms.
```

```
WHERE SaleBase.[Sale Key] = 26974
```

```
Table 'SaleBase'. Scan count 1, logical reads 0, physical reads 0, read-ahead reads 0,
lob logical reads 347, lob physical reads 0,

lob read-ahead reads 0.

Table 'SaleBase'. Segment reads 1, segment skipped 0.

Table 'Worktable'. Scan count 0, logical reads 0, physical reads 0, read-ahead reads 0,
lob logical reads 0, lob physical reads 0,

lob read-ahead reads 0.

Table 'Date'. Scan count 0, logical reads 2, physical reads 0, read-ahead reads 0, lob
logical reads 0, lob physical reads 0,

lob read-ahead reads 0.

Table 'Customer'. Scan count 0, logical reads 2, physical reads 0, read-ahead reads 0,
lob logical reads 0, lob physical reads 0,

lob read-ahead reads 0.

CPU time = 0 ms,  elapsed time = 40 ms.
```

```
CREATE UNIQUE INDEX [Sale Key] ON Fact.SaleBase ([Sale Key]);
CREATE INDEX [WWI Invoice ID] ON Fact.SaleBase ([WWI Invoice ID]);
```

```
Table 'Date'. Scan count 0, logical reads 2, physical reads 0, read-ahead reads 0, lob
logical reads 0, lob physical reads 0,

lob read-ahead reads 0.

Table 'Customer'. Scan count 0, logical reads 2, physical reads 0, read-ahead reads 0,
lob logical reads 0, lob physical reads 0,

lob read-ahead reads 0.

Table 'SaleBase'. Scan count 1, logical reads 3, physical reads 0, read-ahead reads 0,
lob logical reads 186, lob physical reads 0,

lob read-ahead reads 0.

Table 'SaleBase'. Segment reads 1, segment skipped 0.

CPU time = 0 ms,  elapsed time = 30 ms.
```

```
.CREATE TABLE [Fact].[SaleLimited](
        [City Key] [int] NOT NULL,
                [Customer Key] [int] NOT NULL,
        [Bill To Customer Key] [int] NOT NULL,
        [Stock Item Key] [int] NOT NULL,
        [Invoice Date Key] [date] NOT NULL,
        [Delivery Date Key] [date] NULL,
        [Salesperson Key] [int] NOT NULL,
        [WWI Invoice ID] [int] NOT NULL,
        [Description] [nvarchar](100) NOT NULL,
        [Package] [nvarchar](50) NOT NULL,
        [Quantity] [int] NOT NULL
);
```

```
CREATE CLUSTERED COLUMNSTORE INDEX [CColumnStore] ON [Fact].[SaleLimited];
```

```
INSERT INTO [Fact].[SaleLimited] WITH (TABLOCK)
     ([City Key], [Customer Key],  [Bill To Customer Key], [Stock Item Key],
      [Invoice Date Key], [Delivery Date Key],[Salesperson Key],
      [WWI Invoice ID], [Description], [Package], [Quantity])
SELECT TOP (100000) [City Key], [Customer Key],  [Bill To Customer Key],
          [Stock Item Key], [Invoice Date Key], [Delivery Date Key],[Salesperson Key],
          [WWI Invoice ID], [Description], [Package], [Quantity]
FROM Fact.Sale
GO 3 --run this statement 3 times
```

```
SELECT state_desc, total_rows, deleted_rows,
       transition_to_compressed_state_desc as transition
FROM sys.dm_db_column_store_row_group_physical_stats
WHERE object_id  = OBJECT_ID('Fact.SaleLimited');
```

```
state_desc      total_rows deleted_rows transition
--------------- ---------- ------------ ------------------------------
OPEN            150000     0            NULL
OPEN            150000     0            NULL
```

```
ALTER INDEX CColumnStore ON Fact.SaleLimited REORGANIZE
                                    WITH (COMPRESS_ALL_ROW_GROUPS = ON);
```

```
state_desc      total_rows deleted_rows transition
--------------- ---------- ------------ ------------------------------
COMPRESSED      150000     0            REORG_FORCED
COMPRESSED      150000     0            REORG_FORCED
TOMBSTONE       150000     0            NULL
TOMBSTONE       150000     0            NULL
```

```
ALTER INDEX CColumnStore ON Fact.SaleLimited REORGANIZE;
```

```
state_desc      total_rows deleted_rows transition
--------------- ---------- ------------ ------------------------------
COMPRESSED      300000     0            MERGE
TOMBSTONE       150000     0            NULL
TOMBSTONE       150000     0            NULL
```

```
    TRUNCATE TABLE Fact.SaleLimited;
    GO
    INSERT INTO [Fact].[SaleLimited] WITH (TABLOCK)
         ([City Key], [Customer Key],  [Bill To Customer Key], [Stock Item Key],
          [Invoice Date Key], [Delivery Date Key],[Salesperson Key],
          [WWI Invoice ID], [Description], [Package], [Quantity])
    SELECT TOP (102400) [City Key], [Customer Key],  [Bill To Customer Key],
          [Stock Item Key], [Invoice Date Key], [Delivery Date Key],[Salesperson Key],
          [WWI Invoice ID], [Description], [Package], [Quantity]
FROM Fact.Sale
OPTION (MAXDOP 1); --not in parallel
GO 3
```

```
state_desc      total_rows deleted_rows transition
--------------- ---------- ------------ ------------------------------
COMPRESSED      102400     0            BULKLOAD
COMPRESSED      102400     0            BULKLOAD
COMPRESSED      102400     0            BULKLOAD
```

```
state_desc      total_rows deleted_rows transition
--------------- ---------- ------------ ------------------------------
COMPRESSED      307200     0            MERGE
TOMBSTONE       102400     0            NULL
TOMBSTONE       102400     0            NULL
TOMBSTONE       102400     0            NULL
```

```
TRUNCATE TABLE Fact.SaleLimited;
INSERT INTO [Fact].[SaleLimited] WITH (TABLOCK)
([City Key],     [Customer Key],
                 [Bill To Customer Key], [Stock Item Key],
                 [Invoice Date Key], [Delivery Date Key],[Salesperson Key],
                 [WWI Invoice ID], [Description], [Package], [Quantity])

SELECT TOP (5000) [City Key],     [Customer Key],
                 [Bill To Customer Key], [Stock Item Key],
                 [Invoice Date Key], [Delivery Date Key],[Salesperson Key],
                 [WWI Invoice ID], [Description], [Package], [Quantity]

FROM Fact.Sale;
```

```
ALTER INDEX [CColumnStore] ON [Fact].[SaleLimited] REBUILD;
```

```
state_desc      total_rows deleted_rows transition
--------------- ---------- ------------ ------------------------------
COMPRESSED      5000       0            INDEX_BUILD
```

```
TRUNCATE TABLE Fact.SaleLimited;
INSERT INTO [Fact].[SaleLimited]
([City Key],     [Customer Key],
                 [Bill To Customer Key], [Stock Item Key],
                 [Invoice Date Key], [Delivery Date Key],[Salesperson Key],
                 [WWI Invoice ID], [Description], [Package], [Quantity])

SELECT TOP (100000 ) [City Key],  [Customer Key],
                 [Bill To Customer Key], [Stock Item Key],
                 [Invoice Date Key], [Delivery Date Key],[Salesperson Key],
                 [WWI Invoice ID], [Description], [Package], [Quantity]

FROM Fact.Sale;
ALTER INDEX [CColumnStore] ON [Fact].[SaleLimited] REBUILD;
```

```
state_desc      total_rows deleted_rows transition
--------------- ---------- ------------ ------------------------------
COMPRESSED      100000     0            INDEX_BUILD
```

```
DELETE FROM Fact.SaleLimited
WHERE  [Customer Key] = 21;
```

```
state_desc      total_rows deleted_rows transition
--------------- ---------- ------------ ------------------------------
COMPRESSED      100000     135          INDEX_BUILD
```

```
state_desc      total_rows deleted_rows transition
--------------- ---------- ------------ ------------------------------
OPEN            98         0            NULL
COMPRESSED      100000     233          INDEX_BUILD
```

```
SELECT StockItemID, StockItemName, SupplierID, ColorID, UnitPackageID,
       OuterPackageID, Brand, Size
FROM   Examples.Product
WHERE  UnitPackageId = 9;
```

```
CREATE TABLE Sales.InvoiceItemFact
(
    InvoiceItemFactId int NOT NULL
          IDENTITY CONSTRAINT PKInvoiceItemFact PRIMARY KEY,
    ProductDimId int NOT NULL,
    CustomerDimId int NOT NULL,
    DateDimId int NOT NULL,
    SalesAmount money NOT NULL,
    SalesQuantity decimal(10,4)

);
```

```
DisabledFlag bit NOT NULL CONSTRAINT DFTLTableName_DisabledFlag DEFAULT (0);
```

```
CREATE SCHEMA Examples;
GO
CREATE TABLE Examples.Widget
(
    WidgetId    int CONSTRAINT PKWidget PRIMARY KEY,
    RowLastModifiedTime datetime2(0) NOT NULL
);
```

Add the DEFAULT constraint as the default value for the column:

```
ALTER TABLE Examples.Widget
  ADD CONSTRAINT DFLTWidget_RowLastModifiedTime
         DEFAULT (SYSDATETIME()) FOR RowLastModifiedTime;
```

```
INSERT INTO Examples.Widget(WidgetId)
VALUES (1),(2);
INSERT INTO Examples.Widget(WidgetId, RowLastModifiedTime)
VALUES (3,DEFAULT), (4,DEFAULT);
```

```
WidgetId    RowLastModifiedTime
----------- ---------------------------
1           2016-09-14 18:08:28
2           2016-09-14 18:08:28
3           2016-09-14 18:08:28
4           2016-09-14 18:08:28
```

```
UPDATE Examples.Widget
SET RowLastModifiedTime = DEFAULT;
```

```
ALTER TABLE Examples.Widget
  ADD EnabledFlag BIT NOT NULL
      CONSTRAINT DFLTWidget_EnabledFlag DEFAULT (1);
```

```
CREATE TABLE Examples.AllDefaulted
(
    AllDefaultedId int IDENTITY(1,1) NOT NULL,
    RowCreatedTime datetime2(0) NOT NULL
        CONSTRAINT DFLTAllDefaulted_RowCreatedTime DEFAULT (SYSDATETIME()),
    RowModifiedTime datetime2(0) NOT NULL
        CONSTRAINT DFLTAllDefaulted_RowModifiedTime DEFAULT (SYSDATETIME())
);
```

```
INSERT INTO Examples.AllDefaulted
DEFAULT VALUES;
```

```
INSERT INTO Examples.AllDefaulted(RowModifiedTime, RowCreatedTime)
DEFAULT VALUES;
INSERT INTO Examples.AllDefaulted(RowCreatedTime)
DEFAULT VALUES;
```

```
SELECT *
FROM   Examples.AllDefaulted;
```

```
AllDefaultedId RowCreatedTime             RowModifiedTime
-------------- -------------------------- --------------------------
1              2016-09-14 18:19:30        2016-09-14 18:19:30
2              2016-09-14 18:19:30        2016-09-14 18:19:30
3              2016-09-14 18:19:30        2016-09-14 18:19:30
```

```
INSERT INTO Examples.AllDefaulted(AllDefaultedId)
DEFAULT VALUES;
```

```
Msg 339, Level 16, State 1, Line 69
DEFAULT or NULL are not allowed as explicit identity values.
```

```
CREATE TABLE Examples.Gadget
(
    GadgetId    int IDENTITY(1,1) NOT NULL CONSTRAINT PKGadget PRIMARY KEY,
    GadgetCode  varchar(10) NOT NULL
);
```

```
INSERT INTO Examples.Gadget(GadgetCode)
VALUES ('Gadget'), ('Gadget'), ('Gadget');
```

```
DELETE FROM Examples.Gadget WHERE GadgetId in (2,3);

ALTER TABLE Examples.Gadget
   ADD CONSTRAINT AKGadget UNIQUE (GadgetCode);
```

```
INSERT INTO Equipment.Tag(Tag, TagCompanyId)
VALUES ('G001',1);
```

```
Msg 2627, Level 14, State 1, Line 100
Violation of UNIQUE KEY constraint 'AKGadget'. Cannot insert duplicate key in object
'Examples.Gadget'. The duplicate key value is (Gadget).
```

```
CREATE TABLE Examples.GroceryItem
(
   ItemCost smallmoney NULL,
   CONSTRAINT CHKGroceryItem_ItemCostRange
       CHECK (ItemCost > 0 AND ItemCost < 1000)
);
```

```
INSERT INTO Examples.GroceryItem
VALUES (3000.95);
```

```
Msg 547, Level 16, State 0, Line 286
The INSERT statement conflicted with the CHECK constraint
"CHKGroceryItem_ItemCostRange". The conflict occurred in database "ExamBook762Ch2",
table "Examples.GroceryItem", column 'ItemCost'.
```

```
INSERT INTO Examples.GroceryItem
VALUES (100.95);
```

```
INSERT INTO Examples.GroceryItem
VALUES (NULL);
```

```
CREATE TABLE Examples.Message
(
    MessageTag  char(5) NOT NULL,
    Comment nvarchar(max) NULL
);
```

```
ALTER TABLE Examples.Message
   ADD CONSTRAINT CHKMessage_MessageTagFormat
      CHECK (MessageTag LIKE '[A-Z]-[0-9][0-9][0-9]');

ALTER TABLE Examples.Message
   ADD CONSTRAINT CHKMessage_CommentNotEmpty
       CHECK (LEN(Comment) > 0);
```

```
INSERT INTO Examples.Message(MessageTag, Comment)
VALUES ('Bad','');
```

```
Msg 547, Level 16, State 0, Line 312
The INSERT statement conflicted with the CHECK constraint "CHKMessage_MessageTagFormat".
The conflict occurred in database "ExamBook762Ch2", table "Examples.Message",
column 'MessageTag'.
```

```
CREATE TABLE Examples.Customer
(
    ForcedDisabledFlag bit NOT NULL,
    ForcedEnabledFlag bit NOT NULL,
    CONSTRAINT CHKCustomer_ForcedStatusFlagCheck
      CHECK (NOT (ForcedDisabledFlag = 1 AND ForcedEnabledFlag = 1))
);
```

```
CREATE TABLE Examples.Parent
(
    ParentId   int NOT NULL CONSTRAINT PKParent PRIMARY KEY
);
CREATE TABLE Examples.Child
(
    ChildId int NOT NULL CONSTRAINT PKChild PRIMARY KEY,
    ParentId int NULL
);
```

```
ALTER TABLE Examples.Child
     ADD CONSTRAINT FKChild_Ref_ExamplesParent
       FOREIGN KEY (ParentId) REFERENCES Examples.Parent(ParentId);
```

```
INSERT INTO Examples.Parent(ParentId)
VALUES (1),(2),(3);
```

```
INSERT INTO Examples.Child (ChildId, ParentId)
VALUES (1,1);
```

```
INSERT INTO Examples.Child (ChildId, ParentId)
VALUES (2,100);
```

```
Msg 547, Level 16, State 0, Line 124
The INSERT statement conflicted with the FOREIGN KEY constraint
"FKChild_Ref_ExamplesParent". The conflict occurred in database "ExamBook762Ch2",
table "Examples.Parent", column 'ParentId'.
```

```
INSERT INTO Examples.Child (ChildId, ParentId)
VALUES (3,NULL);
```

```
CREATE TABLE Examples.TwoPartKey
(
    KeyColumn1  int NOT NULL,
    KeyColumn2  int NOT NULL,
    CONSTRAINT PKTwoPartKey PRIMARY KEY (KeyColumn1, KeyColumn2)
);

INSERT INTO Examples.TwoPartKey (KeyColumn1, KeyColumn2)

VALUES (1, 1);
CREATE TABLE Examples.TwoPartKeyReference
(
    KeyColumn1 int NULL,
    KeyColumn2 int NULL,
    CONSTRAINT FKTwoPartKeyReference_Ref_ExamplesTwoPartKey
        FOREIGN KEY (KeyColumn1, KeyColumn2)
            REFERENCES Examples.TwoPartKey (KeyColumn1, KeyColumn2)
);
```

```
INSERT INTO Examples.TwoPartKeyReference (KeyColumn1, KeyColumn2)
VALUES (1, 1), (NULL, NULL);
```

```
INSERT INTO Examples.TwoPartKeyReference (KeyColumn1, KeyColumn2)
VALUES (2, 2);
```

```
   Msg 547, Level 16, State 0, Line 157
The INSERT statement conflicted with the FOREIGN KEY constraint "FKTwoPartKeyReference_
Ref_ExamplesTwoPartKey". The conflict occurred in database
"ExamBook762Ch2", table "Examples.TwoPartKey".
```

```
INSERT INTO Examples.TwoPartKeyReference (KeyColumn1, KeyColumn2)
VALUES (6000000, NULL);
```

```
ALTER TABLE Alt.TwoPartKeyReference
      ADD CONSTRAINT CHKTwoPartKeyReference_FKNULLs
           CHECK ((KeyColumn1 IS NULL and KeyColumn2 IS NULL)
                  OR
                  (KeyColumn1 IS NOT NULL and KeyColumn2 IS NOT NULL));
```

```
CREATE TABLE Examples.Invoice
(
    InvoiceId   int NOT NULL CONSTRAINT PKInvoice PRIMARY KEY
);
CREATE TABLE Examples.InvoiceLineItem
(
    InvoiceLineItemId int NOT NULL CONSTRAINT PKInvoiceLineItem PRIMARY KEY,
    InvoiceLineNumber smallint NOT NULL,
    InvoiceId     int NOT NULL
       CONSTRAINT FKInvoiceLineItem_Ref_ExamplesInvoice
          REFERENCES Examples.Invoice(InvoiceId)
            ON DELETE CASCADE
            ON UPDATE NO ACTION,
    CONSTRAINT AKInvoiceLineItem UNIQUE (InvoiceId, InvoiceLineNumber)
);
```

```
INSERT INTO Examples.Invoice(InvoiceId)
VALUES (1),(2),(3);
INSERT INTO Examples.InvoiceLineItem(InvoiceLineItemId, InvoiceId,InvoiceLineNumber)
VALUES (1,1,1),(2,1,2), (3,2,1);
```

```
SELECT Invoice.InvoiceId, InvoiceLineItem.InvoiceLineItemId
FROM   Examples.Invoice
          FULL OUTER JOIN Examples.InvoiceLineItem
            ON Invoice.InvoiceId = InvoiceLineItem.InvoiceId;
```

InvoiceId	InvoiceLineItemId
1	1
1	2
2	3
3	NULL

InvoiceId	InvoiceLineItemId
2	3
3	NULL

```
CREATE TABLE Examples.Code
(
    Code    varchar(10) NOT NULL CONSTRAINT PKCode PRIMARY KEY
);
CREATE TABLE Examples.CodedItem
(
    Code    varchar(10) NOT NULL
         CONSTRAINT FKCodedItem_Ref_ExampleCode
            REFERENCES Examples.Code (Code)
                ON UPDATE CASCADE
);
```

```
INSERT INTO Examples.Code (Code)
VALUES ('Blacke');
INSERT INTO Examples.CodedItem (Code)
VALUES ('Blacke');
```

```
SELECT Code.Code, CodedItem.Code AS CodedItemCode
FROM   Examples.Code
           FULL OUTER JOIN Examples.CodedItem
                ON Code.Code = CodedItem.Code;
```

```
CREATE TABLE Examples.Employee
(
        EmployeeId int NOT NULL CONSTRAINT PKEmployee PRIMARY KEY,
        EmployeeNumber char(8) NOT NULL,
         ManagerId int NULL
               CONSTRAINT FKEmployee_Ref_ExamplesEmployee
                   REFERENCES Examples.Employee (EmployeeId);

 );
```

```
INSERT INTO Examples.Employee(EmployeeId, EmployeeNumber, ManagerId)
VALUES (1,'00000001',NULL), (2,'10000001',1),(3,'10000002',1), (4,'20000001',3);
```

EmployeeId	EmployeeNumber	ManagerId
1	00000001	NULL
2	10000001	1
3	10000002	1
4	20000001	3

```
WITH EmployeeHierarchy AS
(
    SELECT EmployeeID,  CAST(CONCAT('\',EmployeeId,'\') AS varchar(1500)) AS Hierarchy
    FROM HumanResources.Employee
    WHERE ManagedByEmployeeId IS NULL
    UNION ALL
    SELECT Employee.EmployeeID, CAST(CONCAT(Hierarchy,Employee.EmployeeId,'\')
                                                        AS varchar(1500)) AS Hierarchy
    FROM HumanResources.Employee
      INNER JOIN EmployeeHierarchy
        ON Employee.ManagedByEmployeeId = EmployeeHierarchy.EmployeeId
  )
SELECT *
FROM   EmployeeHierarchy;
```

```
CREATE TABLE Examples.Color
(
      ColorId   int NOT NULL CONSTRAINT PKColor PRIMARY KEY,
      ColorName varchar(30) NOT NULL CONSTRAINT AKColor UNIQUE
);
INSERT INTO Examples.Color(ColorId, ColorName)
VALUES (1,'Orange'),(2,'White');
```

```
CREATE TABLE Examples.Product
(
    ProductId int NOT NULL CONSTRAINT PKProduct PRIMARY KEY,
    ColorName varchar(30) NOT NULL
         CONSTRAINT FKProduct_Ref_ExamplesColor
               REFERENCES Examples.Color (ColorName)
);
```

```
INSERT INTO Examples.Product(ProductId,ColorName)
VALUES (1,'Orange');
```

```
INSERT INTO Examples.Product(ProductId,ColorName)
VALUES (2,'Crimson');
```

```
Msg 547, Level 16, State 0, Line 266
The INSERT statement conflicted with the FOREIGN KEY constraint
"FKProduct_Ref_ExamplesColor". The conflict occurred in database "ExamBook762Ch2",
table "Examples.Color", column 'ColorName'.
```

```
CREATE TABLE Examples.Attendee
(
      ShirtSize  varchar(8) NULL
);
```

```
ALTER TABLE Examples. Attendee
    ADD  CONSTRAINT CHKAttendee_ShirtSizeDomain
        CHECK  (ShirtSize in ('S', 'M','L','XL','XXL'));
```

```
INSERT INTO Examples.Attendee(ShirtSize)
VALUES ('LX');
```

```
   Msg 547, Level 16, State 0, Line 346
The INSERT statement conflicted with the CHECK constraint "CHKAttendee _ ShirtSizeDo-
main".
The conflict occurred in database "ExamBook762Ch2", table "Examples.Attendee",
column 'ShirtSize'.
```

```
CREATE TABLE Examples.ShirtSize
(
     ShirtSize varchar(10) NOT NULL CONSTRAINT PKShirtSize PRIMARY KEY
);
INSERT INTO Examples.ShirtSize(ShirtSize)
VALUES ('S'),('M'),('L'),('XL'),('XXL');
```

```
ALTER TABLE Examples.Attendee
    DROP CONSTRAINT CHKAttendee_ShirtSizeDomain;
ALTER TABLE Examples.Attendee
    ADD CONSTRAINT FKAttendee_Ref_ExamplesShirtSize
        FOREIGN KEY (ShirtSize) REFERENCES Examples.ShirtSize(ShirtSize);
```

```
INSERT INTO Examples.Attendee(ShirtSize)
VALUES ('LX');
```

```
Msg 547, Level 16, State 0, Line 364
The INSERT statement conflicted with the FOREIGN KEY constraint
"FKAttendee_Ref_ExamplesShirtSize". The conflict occurred in database "ExamBook762Ch2",
table "Examples.ShirtSize", column 'ShirtSize'.
```

```
CREATE TABLE Examples.CreateTableExample
(
    --Uniqueness constraint referencing single column
    SingleColumnKey int NOT NULL CONSTRAINT PKCreateTableExample PRIMARY KEY,

    --Uniqueness constraint in separate line
    TwoColumnKey1 int NOT NULL,
    TwoColumnKey2 int NOT NULL,
    CONSTRAINT AKCreateTableExample UNIQUE (TwoColumnKey1, TwoColumnKey2),

    --CHECK constraint declare as column constraint
    PositiveInteger int NOT NULL
         CONSTRAINT CHKCreateTableExample_PostiveInteger CHECK (PositiveInteger > 0),

    --CHECK constraint that could reference multiple columns
    NegativeInteger int NOT NULL,
    CONSTRAINT CHKCreateTableExample_NegativeInteger CHECK (NegativeInteger > 0),

    --FOREIGN KEY constraint inline with column
    FKColumn1 int NULL CONSTRAINT FKColumn1_ref_Table REFERENCES Tbl (TblId),

    --FOREIGN KEY constraint… Could reference more than one columns
    FKColumn2 int NULL,
    CONSTRAINT FKColumn2_ref_Table FOREIGN KEY (FKColumn2) REFERENCES Tbl (TblId)
);
```

```
ALTER TABLE Examples.CreateTableExample
    DROP PKCreateTableExample;
ALTER TABLE Examples.CreateTableExample
    ADD CONSTRAINT PKCreateTableExample PRIMARY KEY (SingleColumnKey);
```

```
CREATE TABLE Examples.BadData
(
        PositiveValue int NOT NULL
);
INSERT INTO Examples.BadData(PositiveValue)
VALUES (-1),(-2),(-3),(-4);
```

```
ALTER TABLE Examples.BadData
   ADD CONSTRAINT CHKBadData_PostiveValue CHECK(PositiveValue > 0);
```

```
Msg 547, Level 16, State 0, Line 414
The ALTER TABLE statement conflicted with the CHECK constraint
CHKBadData_PostiveValue". The conflict occurred in database
ExamBook762Ch2", table "Examples.BadData", column 'PositiveValue'.
```

```
ALTER TABLE Examples.BadData WITH NOCHECK
   ADD CONSTRAINT CHKBadData_PostiveValue CHECK(PositiveValue > 0);
```

```
UPDATE Examples.BadData
SET    PositiveValue = PositiveValue;
```

```
Msg 547, Level 16, State 0, Line 420
The UPDATE statement conflicted with the CHECK constraint "CHKBadData_PostiveValue".
The conflict occurred in database "ExamBook762Ch2", table "Examples.BadData",
column 'PositiveValue'.
```

```
DELETE FROM Examples.BadData
WHERE  PositiveValue <= 0;
```

```
SELECT is_not_trusted, is_disabled
FROM   sys.check_constraints --for a FOREIGN KEY, use sys.foreign_keys
WHERE  OBJECT_SCHEMA_NAME(object_id) = 'Examples'
  and  OBJECT_NAME(object_id) = 'CHKBadData_PostiveValue';
```

```
ALTER TABLE Examples.BadData WITH CHECK CHECK
  CONSTRAINT CHKBadData _ PostiveValue;
```

```
ALTER TABLE Examples.BadData
        NOCHECK CONSTRAINT CHKBadData_PostiveValue;
```

```
ALTER TABLE Sales.Invoices
    ADD CONSTRAINT CHKInvoices_OrderIdBetween0and1000000
        CHECK (OrderId BETWEEN 0 AND 1000000);
```

```
CREATE TABLE Examples.ScenarioTestType
(
    ScenarioTestType varchar(10) NOT NULL CONSTRAINT PKScenarioTestType PRIMARY KEY
);
CREATE TABLE Examples.ScenarioTest
(
    ScenarioTestId int NOT NULL PRIMARY KEY,
    ScenarioTestType varchar(10) NULL CONSTRAINT CHKScenarioTest_ScenarioTestType
                                        CHECK ScenarioTestType IN ('Type1','Type2'))

);
ALTER TABLE Examples.ScenarioTest
   ADD CONSTRAINT FKScenarioTest_Ref_ExamplesScenarioTestType
       FOREIGN KEY (ScenarioTestType) REFERENCES Examples.ScenarioTestType;
```

```
INSERT INTO Examples.ScenarioTest(ScenarioTestId, ScenarioTestType)
VALUES (1,'Type1');
INSERT INTO Examples.ScenarioTestType(ScenarioTestType)
VALUES ('Type1');
INSERT INTO Examples.ScenarioTest(ScenarioTestId, ScenarioTestType)
VALUES (1,'Type1');
INSERT INTO Examples.ScenarioTest(ScenarioTestId, ScenarioTestType)
VALUES (1,'Type2');
INSERT INTO Examples.ScenarioTest(ScenarioTestId, ScenarioTestType)
VALUES (2,'Type1');
INSERT INTO Examples.ScenarioTests(ScenarioTestId, ScenarioTestType)
VALUES (3,'Type1');
```

```
INSERT INTO Examples.ScenarioTest(ScenarioTestId, ScenarioTestType)
VALUES (1,'Type1');
```

```
INSERT INTO Examples.ScenarioTestType(ScenarioTestType)
VALUES ('Type 1')
```

```
INSERT INTO Examples.ScenarioTest(ScenarioTestId, ScenarioTestType)
VALUES (1,'Type1');
```

```
INSERT INTO Examples.ScenarioTest(ScenarioTestId, ScenarioTestType)
VALUES (1,'Type2');
```

```
INSERT INTO Examples.ScenarioTest(ScenarioTestId, ScenarioTestType)
VALUES (2,'Type1');
```

```
INSERT INTO Examples.ScenarioTests(ScenarioTestId, ScenarioTestType)
VALUES (3,'Type1');
```

```
CREATE SCHEMA Examples;
GO
CREATE TABLE Examples.Company
(
    CompanyName   nvarchar(50) NOT NULL CONSTRAINT PKCompany PRIMARY KEY,
    CompanyURL nvarchar(max) NOT NULL
);
```

```
INSERT INTO Examples.Company(CompanyName, CompanyURL)
VALUES ('Blue Yonder Airlines','http://www.blueyonderairlines.com/'),
       ('Tailspin Toys','http://www.tailspintoys.com/');
```

```
CompanyName                  CompanyURL
---------------------------- ------------------------------------
Blue Yonder Airlines         http://www.blueyonderairlines.com/
Tailspin Toys                http://www.tailspintoys.com/
```

```
DROP TABLE IF EXISTS Examples.Company;
CREATE TABLE Examples.Company
(
    CompanyId     int NOT NULL IDENTITY(1,1) CONSTRAINT PKCompany PRIMARY KEY,
    CompanyName   nvarchar(50) NOT NULL CONSTRAINT AKCompany UNIQUE,
    CompanyURL nvarchar(max) NOT NULL
);
```

```
INSERT INTO Examples.Company(CompanyName, CompanyURL)
VALUES ('Blue Yonder Airlines','http://www.blueyonderairlines.com/'),
       ('Tailspin Toys','http://www.tailspintoys.com/');
```

```
INSERT INTO Examples.Company(CompanyName, CompanyURL)
VALUES ('Blue Yonder Airlines','http://www.blueyonderairlines.com/');
```

```
Msg 2627, Level 14, State 1, Line 53
Violation of UNIQUE KEY constraint 'AKCompany'. Cannot insert duplicate key in object
'Examples.Company'. The duplicate key value is (Blue Yonder Airlines).
```

```
INSERT INTO Examples.Company(CompanyName, CompanyURL)
VALUES ('Northwind Traders','http://www.northwindtraders.com/');
```

```
CompanyId   CompanyName           CompanyURL
----------- --------------------- ---------------------------------------
1           Blue Yonder Airlines  http://www.blueyonderairlines.com/
2           Tailspin Toys         http://www.tailspintoys.com/
4           Northwind Traders     http://www.northwindtraders.com/
```

```
DROP TABLE IF EXISTS Examples.Company;
DROP SEQUENCE IF EXISTS Examples.Company_SEQUENCE;

CREATE SEQUENCE Examples.Company_SEQUENCE AS INT START WITH 1;
CREATE TABLE Examples.Company
(
    CompanyId     int NOT NULL CONSTRAINT PKCompany PRIMARY KEY
                               CONSTRAINT DFLTCompany_CompanyId DEFAULT
                                          (NEXT VALUE FOR Examples.Company_SEQUENCE),
    CompanyName   nvarchar(50) NOT NULL CONSTRAINT AKCompany UNIQUE,
    CompanyURL nvarchar(max) NOT NULL
);
```

```
INSERT INTO Examples.Company(CompanyName, CompanyURL)
VALUES ('Blue Yonder Airlines','http://www.blueyonderairlines.com/'),
       ('Tailspin Toys','http://www.tailspintoys.com/');
```

```
DECLARE @CompanyId INT = NEXT VALUE FOR Examples.Company_SEQUENCE;


INSERT INTO Examples.Company(CompanyId, CompanyName, CompanyURL)
VALUES (@CompanyId, 'Northwind Traders','http://www.northwindtraders.com/');
```

```
DROP TABLE IF EXISTS Examples.Company;
CREATE TABLE Examples.Company
(
    CompanyId     uniqueidentifier NOT NULL CONSTRAINT PKCompany PRIMARY KEY
                                CONSTRAINT DFLTCompany_CompanyId DEFAULT (NEWID()),
    CompanyName   nvarchar(50) NOT NULL CONSTRAINT AKCompany UNIQUE,
    CompanyURL nvarchar(max) NOT NULL
);
```

```
CREATE TABLE Examples.DriversLicense
(
        Locality char(10) NOT NULL,
        LicenseNumber varchar(40) NOT NULL,
        CONSTRAINT PKDriversLicense PRIMARY KEY (Locality, LicenseNumber)

);
```

```
CREATE TABLE Examples.EmployeeDriverLicense
(
        EmployeeNumber char(10) NOT NULL, --Ref to Employee table
        Locality char(10) NOT NULL, --Ref to DriversLicense table
        LicenseNumber varchar(40) NOT NULL, --Ref to DriversLicense table
        CONSTRAINT PKEmployeeDriversLicense PRIMARY KEY
                  (EmployeeNumber, Locality, LicenseNumber)
);
```

```
CREATE TABLE Examples.DriversLicense
(
        DriversLicenseId int CONSTRAINT PKDriversLicense PRIMARY KEY,
        Locality char(10) NOT NULL,
        LicenseNumber varchar(40) NOT NULL,
        CONSTRAINT AKDriversLicense UNIQUE (Locality, LicenseNumber)
);
CREATE TABLE Examples.EmployeeDriverLicense
(
        EmployeeDriverLicenseId int NOT NULL
                CONSTRAINT PKEmployeeDriverLicense PRIMARY KEY,
        EmployeeId int NOT NULL, --Ref to Employee table
        DriversLicenseId int NOT NULL, --Ref to DriversLicense table
        CONSTRAINT AKEmployeeDriverLicense UNIQUE (EmployeeId, DriversLicenseId)
);
```

```
CREATE PROCEDURE SchemaName.ObjectName

[WITH options]
[FOR REPLICATION]
    @Parameter1 datatype,
    @Parameter2 datatype = 'Optional Default',
    @Parameter3 datatype = NULL
AS
    1 or more Transact-SQL statements;
```

```
CREATE TABLE Examples.SimpleTable
(
    SimpleTableId int NOT NULL IDENTITY(1,1)
                CONSTRAINT PKSimpleTable PRIMARY KEY,
    Value1   varchar(20) NOT NULL,
    Value2   varchar(20) NOT NULL
);
```

```
CREATE PROCEDURE Examples.SimpleTable_Insert
    @SimpleTableId int,
    @Value1  varchar(20),
    @Value2  varchar(20)
AS
    INSERT INTO Examples.SimpleTable(Value1, Value2)
    VALUES (@Value1, @Value2);
GO

CREATE PROCEDURE Examples.SimpleTable_Update
    @SimpleTableId int,
    @Value1  varchar(20),
    @Value2  varchar(20)
AS
    UPDATE Examples.SimpleTable
    SET Value1 = @Value1,
        Value2 = @Value2
    WHERE SimpleTableId = @SimpleTableId;
GO
CREATE PROCEDURE Examples.SimpleTable_Delete
    @SimpleTableId int,
    @Value  varchar(20)
AS
    DELETE Examples.SimpleTable
    WHERE SimpleTableId = @SimpleTableId
GO
```

```
CREATE PROCEDURE Examples.SimpleTable_Select
AS
    SELECT SimpleTableId, Value1, Value2
    FROM Examples.SimpleTable
    ORDER BY Value1;
```

```
CREATE PROCEDURE Examples.SimpleTable_SelectValue1StartWithQorZ
AS
   SELECT SimpleTableId, Value1, Value2
   FROM Examples.SimpleTable
   WHERE Value1 LIKE 'Q%'
   ORDER BY Value1;

   SELECT SimpleTableId, Value1, Value2
   FROM Examples.SimpleTable
   WHERE Value1 LIKE 'Z%'
   ORDER BY Value1 DESC;
```

```
CREATE PROCEDURE Examples.SimpleTable_SelectValue1StartWithQorZ
AS
  IF DATENAME(weekday,getdate()) NOT IN ('Saturday','Sunday')
     SELECT SimpleTableId, Value1, Value2
     FROM Examples.SimpleTable
     WHERE  Value1 LIKE '[QZ]%';
```

```
CREATE PROCEDURE Examples.ProcedureName
AS
SELECT ColumnName From Bogus.TableName;
```

```
CREATE PROCEDURE Examples.SimpleTable_Select
AS
    SET NOCOUNT ON;
    SELECT SimpleTableId, Value1, Value2
    FROM Examples.SimpleTable
    ORDER BY Value1;
```

```
CREATE TABLE Examples.Parameter
(
    ParameterId int NOT NULL IDENTITY(1,1) CONSTRAINT PKParameter PRIMARY KEY,
    Value1   varchar(20) NOT NULL,
    Value2  varchar(20) NOT NULL,
)
```

```
CREATE PROCEDURE Examples.Parameter_Insert
    @Value1 varchar(20) = 'No entry given',
    @Value2 varchar(20) = 'No entry given'
AS
    SET NOCOUNT ON;
    INSERT INTO Examples.Parameter(Value1,Value2)
    VALUES (@Value1, @Value2);
```

```
--using all defaults
EXECUTE Examples.Parameter_Insert;

--by position, @Value1 parameter only
EXECUTE Examples.Parameter_Insert 'Some Entry';

--both columns by position
EXECUTE Examples.Parameter_Insert 'More Entry','More Entry';

-- using the name of the parameter (could also include @Value2);
EXECUTE Examples.Parameter_Insert @Value1 = 'Other Entry';

--starting positionally, but finishing by name
EXECUTE Examples.Parameter_Insert 'Mixed Entry', @Value2 = 'Mixed Entry';
```

```
EXECUTE Examples.Parameter_Insert @Value1 = 'Remixed Entry', 'Remixed Entry';
```

```
Msg 119, Level 15, State 1, Line 736
Must pass parameter number 2 and subsequent parameters as '@name = value'. After the
form '@name = value' has been used, all subsequent parameters must be passed in the form
'@name = value'.
```

```
ALTER PROCEDURE Examples.Parameter_Insert
    @Value1 varchar(20) = 'No entry given',
    @Value2 varchar(20) = 'No entry given' OUTPUT,
    @NewParameterId int = NULL OUTPUT
AS
    SET NOCOUNT ON;
    SET @Value1 = UPPER(@Value1);
    SET @Value2 = LOWER(@Value2);

    INSERT INTO Examples.Parameter(Value1,Value2)
    VALUES (@Value1, @Value2);

    SET @NewParameterId = SCOPE_IDENTITY();
```

```
DECLARE @Value1 varchar(20) = 'Test',
        @Value2 varchar(20) = 'Test',
        @NewParameterId int = -200;

EXECUTE Examples.Parameter_Insert @Value1 = @Value1,
                                  @Value2 = @Value2 OUTPUT,
                                  @NewParameterId = @NewParameterId OUTPUT;

SELECT @Value1 as Value1, @Value2 as Value2, @NewParameterId as NewParameterId;

SELECT *
FROM Examples.Parameter
WHERE ParameterId = @newParameterId;
```

```
Value1               Value2               NewParameterId
-------------------- -------------------- --------------
Test                 test                 7

ParameterId Value1               Value2
----------- -------------------- --------------------
7           TEST                 test
```

```
CREATE TABLE Examples.Machine
(
    MachineId   int NOT NULL CONSTRAINT PKMachine PRIMARY KEY,
    MachineNumber char(3) NOT NULL CONSTRAINT AKMachine UNIQUE,
    Description varchar(50) NOT NULL
);
INSERT INTO Examples.Machine(MachineId, MachineNumber, Description)
VALUES (1,'001','Thing1'),(2,'002','Thing2'),(3,'003','Thing3');
```

```
CREATE PROCEDURE Examples.Machine_MultiSelect
    @MachineList varchar(200)
AS
    SET NOCOUNT ON;
    SELECT Machine.MachineId, Machine.MachineNumber
    FROM   Examples.Machine
            JOIN STRING_SPLIT(@MachineList,',') AS StringList
                ON StringList.value = Machine.MachineId;
```

```
CREATE TYPE Examples.SurrogateKeyList AS table
(
    SurrogateKeyId int PRIMARY KEY --note you cannot name constraints for table types

);
```

```
ALTER PROCEDURE Examples.Machine_MultiSelect
    @MachineList Examples.SurrogateKeyList READONLY
AS
    SET NOCOUNT ON;
    SELECT Machine.MachineId, Machine.MachineNumber
    FROM   Examples.Machine
            JOIN @MachineList AS MachineList
                ON MachineList.SurrogateKeyId = Machine.MachineId;
```

```
DECLARE @MachineList Examples.SurrogateKeyList;
INSERT INTO @MachineList (SurrogateKeyId)
VALUES (1),(3);

EXECUTE Examples.Machine_MultiSelect @MachineList = @MachineList;
```

```
CREATE TYPE Examples.Machine AS TABLE
(
    MachineId int NOT NULL PRIMARY KEY,
    MachineNumber char(3) NOT NULL UNIQUE,
    Description varchar(50) NOT NULL
);
```

```
CREATE PROCEDURE Examples.Machine_MultiInsert
    @MachineList Examples.Machine READONLY
AS
    SET NOCOUNT ON;
    INSERT INTO Examples.Machine(MachineId, MachineNumber, Description)
    SELECT MachineId, MachineNumber, Description
    FROM   @MachineList;
```

```
DECLARE @NewMachineRows Examples.Machine;
INSERT INTO @NewMachineRows (MachineId, MachineNumber, Description)
VALUES (4,'004','NewThing4'), (5, '005','NewThing5');

EXECUTE Examples.Machine_MultiInsert @MachineList = @NewMachineRows;
```

```
CREATE PROCEDURE SimpleReturnValue
AS
    DECLARE @NoOp int;
```

```
DECLARE @ReturnCode int;
EXECUTE @ReturnCode = SimpleReturnValue;
SELECT @ReturnCode as ReturnCode;
```

```
CREATE PROCEDURE DoOperation
(
    @Value  int
)
--Procedure returns via return code:
-- 1 - successful execution, with 0 entered
-- 0 - successful execution
-- -1 - invalid, NULL input
AS
    IF @Value = 0
        RETURN 1;
    ELSE IF @Value IS NULL
        RETURN -1;
    ElSE RETURN 0;
```

```
DECLARE @ReturnCode int;
EXECUTE @ReturnCode = DoOperation @Value = NULL;
SELECT  @ReturnCode,
        CASE @ReturnCode WHEN 1 THEN 'Success, 0 Entered'
                         WHEN -1 THEN 'Invalid Input'
                         WHEN 0 THEN 'Success'
        END as ReturnMeaning;
```

Return	Meaning
-1	Invalid Input

```
CREATE TABLE Examples.Player
(
    PlayerId    int NOT NULL CONSTRAINT PKPlayer PRIMARY KEY,
    TeamId      int NOT NULL, --not implemented reference to Team Table
    PlayerNumber char(2) NOT NULL,
    CONSTRAINT AKPlayer UNIQUE (TeamId, PlayerNumber)
)
INSERT INTO Examples.Player(PlayerId, TeamId, PlayerNumber)
VALUES (1,1,'18'),(2,1,'45'),(3,1,'40');
```

```
CREATE PROCEDURE Examples.Player_GetByPlayerNumber
(
    @PlayerNumber char(2)
) AS
    SET NOCOUNT ON;
    DECLARE @PlayerList TABLE (PlayerId int NOT NULL);

    DECLARE @Cursor cursor,
            @Loop_PlayerId int,
            @Loop_PlayerNumber char(2)

    SET @cursor = CURSOR FAST_FORWARD FOR ( SELECT PlayerId, PlayerNumber
                                            FROM   Examples.Player);

    OPEN @cursor;
    WHILE (1=1)
      BEGIN
            FETCH NEXT FROM @Cursor INTO @Loop_PlayerId, @Loop_PlayerNumber
            IF @@FETCH_STATUS <> 0
                BREAK;

            IF @PlayerNumber = @Loop_PlayerNumber
                INSERT INTO @PlayerList(PlayerId)
                VALUES (@Loop_PlayerId);

             END;

          SELECT Player.PlayerId, Player.TeamId
          FROM   Examples.Player
                   JOIN @PlayerList AS PlayerList
                      on PlayerList.PlayerId = Player.PlayerId;
```

```
EXECUTE  Examples.Player_Get @PlayerNumber = '18';
```

```
ALTER PROCEDURE Examples.Player_GetByPlayerNumber
(
    @PlayerNumber char(2)
) AS
    SET NOCOUNT ON

    SELECT Player.PlayerId, Player.TeamId
    FROM   Examples.Player
    WHERE  PlayerNumber = @PlayerNumber;
```

```
EXECUTE  Examples.Player_GetByPlayerNumber @PlayerNumber = '18';
```

```
CREATE TABLE Game
…
GameStartTime  datetime2(0)
…
CREATE PROCEDURE Game_GetForDate
(
        @SearchDate date
…
      FROM  Game
      WHERE CAST(GameTime AS DATE) = @SearchDate;
```

```
FROM GAME
WHERE GameTime >= @SearchDate
    AND   GameTime < DATEADD(Day, 1, @SearchDate);
```

```
CREATE TABLE Order
…
    OrderNumber nvarchar(10)

…

CREATE PROCEDURE Order_Search
@OrderNumber int --because a "number" is an integer, one surmises
…
WHERE OrderNumber = @OrderNumber;
```

```
THROW 50000, 'This is an error message',1;
```

```
Msg 50000, Level 16, State 1, Line 1115
This is an error message
```

```
RAISERROR ('This is an error message',16,1);
```

```
THROW 50000, 'This is an error message',1;
SELECT 'Batch continued'
```

```
Msg 50000, Level 16, State 1, Line 1117
This is an error message
```

```
RAISERROR ('This is an error message',16,1);
SELECT 'Batch continued'
```

```
Msg 50000, Level 16, State 1, Line 1119
This is an error message

----------------
Batch continued
```

```
CREATE PROCEDURE DoOperation
(
    @Value  int
)
AS
    SET NOCOUNT ON;
    IF @Value = 0
        RETURN 1;
    ELSE IF @Value IS NULL
     BEGIN
        THROW 50000, 'The @value parameter should not be NULL',1;
        SELECT 'Continued to here';
        RETURN -1;
     END
    ELSE RETURN 0;
```

```
DECLARE @ReturnCode int
EXECUTE @ReturnCode = DoOperation @Value = NULL;
SELECT  @ReturnCode AS ReturnCode;
```

```
Msg 50000, Level 16, State 1, Procedure DoOperation, Line 10
The @value parameter should not be NULL
```

```
RAISERROR ('The @value parameter should not be NULL',16,1);
```

```
Msg 50000, Level 16, State 1, Procedure DoOperation, Line 11
The @value parameter should not be NULL

-----------------
Continued to here

-----------
-1
```

```
CREATE TABLE Examples.ErrorTesting
(
    ErrorTestingId int NOT NULL CONSTRAINT PKErrorTesting PRIMARY KEY,
    PositiveInteger int NOT NULL
         CONSTRAINT CHKErrorTesting_PositiveInteger CHECK PositiveInteger > 0)
);
```

```
INSERT INTO Examples.ErrorTesting(ErrorTestingId, PositiveInteger)
VALUES (1,1); --Succeed
INSERT INTO Examples.ErrorTesting(ErrorTestingId, PositiveInteger)
VALUES (1,1); --Fail PRIMARY KEY violation
INSERT INTO Examples.ErrorTesting(ErrorTestingId, PositiveInteger)
VALUES (2,-1); --Fail CHECK constraint violation
INSERT INTO Examples.ErrorTesting(ErrorTestingId, PositiveInteger)
VALUES (2,2); --Succeed
SELECT *
FROM   Examples.ErrorTesting;
```

```
Msg 2627, Level 14, State 1, Line 1113
Violation of PRIMARY KEY constraint 'PKErrorTesting'. Cannot insert duplicate key
in object 'Examples.ErrorTesting'. The duplicate key value is (1).

Msg 547, Level 16, State 0, Line 1116
The INSERT statement conflicted with the CHECK constraint
"CHKErrorTesting_PositiveInteger". The conflict occurred in database
"ExamBook762Ch2", table "Examples.ErrorTesting", column 'PositiveInteger'.

ErrorTestingId PositiveInteger
-------------- ---------------
1              1
2              2
```

```
CREATE PROCEDURE Examples.ErrorTesting_InsertTwo
AS
    SET NOCOUNT ON;
    INSERT INTO Examples.ErrorTesting(ErrorTestingId, PositiveInteger)
    VALUES (3,3); --Succeeds

    IF @@ERROR <> 0
       BEGIN
            THROW 50000, 'First statement failed', 1;
            RETURN -1;
       END;

    INSERT INTO Examples.ErrorTesting(ErrorTestingId, PositiveInteger)
    VALUES (4,-1); --Fail Constraint

    IF @@ERROR <> 0
       BEGIN
            THROW 50000, 'Second statement failed', 1;
            RETURN -1;
       END;

          INSERT INTO Examples.ErrorTesting(ErrorTestingId, PositiveInteger)
          VALUES (5,1); --Will succeed if statement executes
          IF @@ERROR <> 0
BEGIN
     THROW 50000, 'Third statement failed', 1;
     RETURN -1;
END;
```

```
EXECUTE Examples.ErrorTesting_InsertTwo;
```

```
Msg 547, Level 16, State 0, Procedure ErrorTesting_InsertTwo, Line 12
The INSERT statement conflicted with the CHECK constraint
"CHKErrorTesting_PositiveInteger". The conflict occurred in database
"ExamBook762Ch2", table "Examples.ErrorTesting", column 'PositiveInteger'.

Msg 50000, Level 16, State 1, Procedure ErrorTesting_InsertTwo, Line 17
Second statement failed
```

```
BEGIN TRY
    --Code you want to execute
END TRY
BEGIN CATCH
    --What to do if the code fails
END CATCH;
```

```
ALTER PROCEDURE Examples.ErrorTesting_InsertTwo
AS
    SET NOCOUNT ON;
    DECLARE @Location nvarchar(30);

    BEGIN TRY
        SET @Location = 'First statement';
        INSERT INTO Examples.ErrorTesting(ErrorTestingId, PositiveInteger)
        VALUES (6,3); --Succeeds

        SET @Location = 'Second statement';
        INSERT INTO Examples.ErrorTesting(ErrorTestingId, PositiveInteger)
        VALUES (7,-1); --Fail Constraint

        SET @Location = 'First statement';
        INSERT INTO Examples.ErrorTesting(ErrorTestingId, PositiveInteger)
        VALUES (8,1); --Will succeed if statement executes
    END TRY
    BEGIN CATCH
        SELECT ERROR_PROCEDURE() AS ErrorProcedure, @Location AS ErrorLocation
        SELECT ERROR_MESSAGE() as ErrorMessage;
        SELECT ERROR_NUMBER() AS ErrorNumber, ERROR_SEVERITY() as ErrorSeverity,
               ERROR_LINE() As ErrorLine;

              THROW;

          END CATCH;
```

```
EXECUTE Examples.ErrorTesting_InsertTwo;
```

```
ErrorProcedure                  ErrorLocation
------------------------------- ------------------------------
ErrorTesting_InsertTwo          Second statement

ErrorMessage
-------------------------------------------------------------------------------
The INSERT statement conflicted with the CHECK constraint
"CHKErrorTesting_PositiveInteger". The conflict occurred in database "ExamBook762Ch2",
table "Examples.ErrorTesting", column 'PositiveInteger'.

ErrorNumber ErrorSeverity ErrorLine
----------- ------------- -----------
547         16            12

Msg 547, Level 16, State 0, Procedure ErrorTesting_InsertTwo, Line 12
The INSERT statement conflicted with the CHECK constraint
"CHKErrorTesting_PositiveInteger". The conflict occurred in database "ExamBook762Ch2",
table "Examples.ErrorTesting", column 'PositiveInteger'.
```

```
BEGIN TRANSACTION;
INSERT INTO Examples.ErrorTesting(ErrorTestingId, PositiveInteger)
VALUES (9,1);

BEGIN TRANSACTION;
SELECT * FROM Examples.ErrorTesting WHERE ErrorTestingId = 9;
ROLLBACK TRANSACTION;
SELECT * FROM Examples.ErrorTesting WHERE ErrorTestingId = 9;
```

```
CREATE TABLE Examples.Worker
(
    WorkerId int NOT NULL IDENTITY(1,1) CONSTRAINT PKWorker PRIMARY KEY,
    WorkerName nvarchar(50) NOT NULL CONSTRAINT AKWorker UNIQUE
);
CREATE TABLE Examples.WorkerAssignment
(
    WorkerAssignmentId int IDENTITY(1,1) CONSTRAINT PKWorkerAssignment PRIMARY KEY,
    WorkerId int NOT NULL,
    CompanyName nvarchar(50) NOT NULL
       CONSTRAINT CHKWorkerAssignment_CompanyName
           CHECK (CompanyName <> 'Contoso, Ltd.'),
    CONSTRAINT AKWorkerAssignment UNIQUE (WorkerId, CompanyName)
);
```

```
INSERT INTO Examples.Worker...
INSERT INTO Examples.WorkerAssignment...
```

```
CREATE PROCEDURE Examples.Worker_AddWithAssignment
    @WorkerName nvarchar(50),
    @CompanyName nvarchar(50)
AS
    SET NOCOUNT ON;
    --do any non-data testing before starting the transaction
    IF @WorkerName IS NULL or @CompanyName IS NULL
        THROW 50000,'Both parameters must be not null',1;

    DECLARE @Location nvarchar(30), @NewWorkerId int;
    BEGIN TRY
        BEGIN TRANSACTION;

        SET @Location = 'Creating Worker Row';
        INSERT INTO Examples.Worker(WorkerName)
        VALUES (@WorkerName);

        SELECT @NewWorkerId = SCOPE_IDENTITY(),
               @Location = 'Creating WorkAssignment Row';

        INSERT INTO Examples.WorkerAssignment(WorkerId, CompanyName)
        VALUES (@NewWorkerId, @CompanyName);

        COMMIT TRANSACTION;
    END TRY
    BEGIN CATCH
        --at the end of the call, we want the transaction rolled back
        --rollback the transaction first, so it definitely occurs as the THROW
        --statement would keep it from happening.
        IF XACT_STATE() <> 0 --if there is a transaction in effect
                             --commitable or not
            ROLLBACK TRANSACTION;

        --format a message that tells the error and then THROW it.
        DECLARE @ErrorMessage nvarchar(4000);
        SET @ErrorMessage = CONCAT('Error occurred during: ''',@Location,'''',
                                   ' System Error: ',
                                   ERROR_NUMBER(),':',ERROR_MESSAGE());
        THROW 50000, @ErrorMessage, 1;
    END CATCH;
```

```
EXEC Examples.Worker_AddWithAssignment @WorkerName = NULL, @CompanyName = NULL;
```

```
Msg 50000, Level 16, State 1, Procedure Worker_AddWithAssignment, Line 7
Both parameters must be not null
```

```
EXEC Examples.Worker_AddWithAssignment
                        @WorkerName='David So', @CompanyName='Margie''s Travel';
```

```
EXEC Examples.Worker_AddWithAssignment
                        @WorkerName='David So', @CompanyName='Margie''s Travel';
```

```
Msg 50000, Level 16, State 1, Procedure Worker_AddWithAssignment, Line 38
Error occurred during: 'Creating Worker Row' System Error: 2627:Violation of UNIQUE KEY
constraint 'AKWorker'. Cannot insert duplicate key in object 'Examples.Worker'. The
duplicate key value is (David So).
```

```
EXEC Examples.Worker_AddWithAssignment
                        @WorkerName='Ian Palangio', @CompanyName='Contoso, Ltd.';
```

```
Msg 50000, Level 16, State 1, Procedure Worker_AddWithAssignment, Line 38
Error occurred during: 'Creating WorkAssignment Row' System Error: 547:The INSERT
statement conflicted with the CHECK constraint "CHKWorkerAssignment_CompanyName". The
conflict occurred in database "ExamBook762Ch2", table "Examples.WorkerAssignment",
column 'CompanyName'.
```

```
EXEC Examples.Worker_AddWithAssignment
                    @WorkerName='Ian Palangio', @CompanyName='Humongous Insurance';
```

```
ALTER PROCEDURE Examples.Worker_AddWithAssignment
    @WorkerName nvarchar(50),
    @CompanyName nvarchar(50)
AS
    SET NOCOUNT ON;
DECLARE @NewWorkerId int;
--still check the parameter values first
IF @WorkerName IS NULL or @CompanyName IS NULL
    THROW 50000,'Both parameters must be not null',1;
--Start a transaction
BEGIN TRANSACTION
INSERT INTO Examples.Worker(WorkerName)
VALUES (@WorkerName);
--check the value of the @@error system function
IF @@ERROR <> 0
  BEGIN
    --rollback the transaction before the THROW (or RETURN if using), because
    --otherwise the THROW will end the batch and transaction stay open
    ROLLBACK TRANSACTION;
    THROW 50000,'Error occurred inserting data into Examples.Worker table',1;
  END;
SELECT @NewWorkerId = SCOPE_IDENTITY()

INSERT INTO Examples.WorkerAssignment(WorkerId, CompanyName)
VALUES (@NewWorkerId, @CompanyName);
 IF @@ERROR <> 0
  BEGIN
    ROLLBACK TRANSACTION;
    THROW 50000,
      'Error occurred inserting data into Examples.WorkerAssignment table',1;
  END;
--if you get this far in the batch, you can commit the transaction
COMMIT TRANSACTION;
```

```
EXEC Examples.Worker_AddWithAssignment @WorkerName='Seth Grossman', @
CompanyName='Margie''s Travel';

GO
--Cause an error due to duplicating all of the data from previous call
EXEC Examples.Worker_AddWithAssignment @WorkerName='Seth Grossman', @
CompanyName='Margie''s Travel';
```

```
Msg 2627, Level 14, State 1, Procedure Worker_AddWithAssignment, Line 14
Violation of UNIQUE KEY constraint 'AKWorker'. Cannot insert duplicate key in object
'Examples.Worker'. The duplicate key value is (Seth Grossman).

Msg 50000, Level 16, State 1, Procedure Worker_AddWithAssignment, Line 21
Error occurred inserting data into Examples.Worker table
```

```
ALTER PROCEDURE Examples.Worker_AddWithAssignment
    @WorkerName nvarchar(50),
    @CompanyName nvarchar(50)
AS
    SET NOCOUNT ON;
    --will cause batch to end on any error
    SET XACT_ABORT ON;

    DECLARE @NewWorkerId int;

    --Same parameter check as other cases
    IF @WorkerName IS NULL or @CompanyName IS NULL
        THROW 50000,'Both parameters must be not null',1;

    --start the transaction
    BEGIN TRANSACTION;
    --  Execute the code as normal
    INSERT INTO Examples.Worker(WorkerName)
    VALUES (@WorkerName);

    SELECT @NewWorkerId = SCOPE_IDENTITY()

    INSERT INTO Examples.WorkerAssignment(WorkerId, CompanyName)
    VALUES (@NewWorkerId, @CompanyName);

    COMMIT TRANSACTION;
```

```
EXEC Examples.Worker_AddWithAssignment
            @WorkerName='Stig Panduro', @CompanyName='Margie''s Travel';

GO
--Cause an error due to duplicating all of the data from previous call
EXEC Examples.Worker_AddWithAssignment
            @WorkerName='Stig Panduro', @CompanyName='Margie''s Travel';
```

```
Msg 2627, Level 14, State 1, Procedure Worker_AddWithAssignment, Line 12
Violation of UNIQUE KEY constraint 'AKWorker'. Cannot insert duplicate key in object
'Examples.Worker'. The duplicate key value is (Stig Panduro).
```

```
CREATE PROCEDURE ChangeTransactionLevel
AS
    BEGIN TRANSACTION;
    ROLLBACK TRANSACTION;
```

```
Msg 266, Level 16, State 2, Procedure ChangeTransactionLevel, Line 0
Transaction count after EXECUTE indicates a mismatching number of BEGIN and COMMIT
statements. Previous count = 1, current count = 0.
Msg 3903, Level 16, State 1, Line 1434
The ROLLBACK TRANSACTION request has no corresponding BEGIN TRANSACTION.
```

```
ALTER PROCEDURE ChangeTransactionLevel
AS
    BEGIN TRANSACTION;
    ROLLBACK TRANSACTION;
    THROW 50000,'Error After Rollback',1;
```

```
Msg 50000, Level 16, State 1, Procedure ChangeTransactionLevel, Line 5
Error After Rollback
```

```
ALTER PROCEDURE dbo.CallChangeTransactionLevel
AS
    BEGIN TRY
        BEGIN TRANSACTION

        DECLARE @Location nvarchar(30) = 'Execute Procedure';
        EXECUTE ChangeTransactionLevel; --This will cause an error by design

        COMMIT TRANSACTION
    END TRY
    BEGIN CATCH
        IF XACT_STATE() <> 0
            ROLLBACK;
        DECLARE @ErrorMessage nvarchar(4000)
        SET @ErrorMessage = CONCAT('Error occurred during: ''',@Location,'''',
                                   ' System Error: ',ERROR_NUMBER(),':',
                                   ERROR_MESSAGE());
        THROW 50000, @ErrorMessage, 1;
    END CATCH;
```

```
EXECUTE dbo.CallChangeTransactionLevel;
```

```
Msg 50000, Level 16, State 1, Procedure CallChangeTransactionLevel, Line 19
Error occurred during: 'Execute Procedure' System Error: 50000:Error After Rollback
```

```
CREATE TABLE Examples.AccountContact
(
     AccountContactId int NOT NULL CONSTRAINT PKAccountContact PRIMARY KEY,
     AccountId        char(4) NOT NULL,
     PrimaryContactFlag bit NOT NULL
);
```

```
SELECT AccountId, SUM(CASE WHEN PrimaryContactFlag = 1 THEN 1 ELSE 0 END)
FROM   Examples.AccountContact
GROUP BY AccountId
HAVING SUM(CASE WHEN PrimaryContactFlag = 1 THEN 1 ELSE 0 END) <> 1;
```

```
CREATE TRIGGER Examples.AccountContact_TriggerAfterInsertUpdate
ON Examples.AccountContact
AFTER INSERT, UPDATE AS
BEGIN
  SET NOCOUNT ON;
  SET ROWCOUNT 0; --in case the client has modified the rowcount
  BEGIN TRY
  --check to see if data is returned by the query from previously
  IF EXISTS ( SELECT AccountId
              FROM   Examples.AccountContact
                     --correlates the changed rows in inserted to the other rows
                     --for the account, so we can check if the rows have changed
              WHERE  EXISTS (SELECT *
                             FROM   inserted
                             WHERE  inserted.AccountId =
                                                 AccountContact.AccountId

                             UNION ALL
                             SELECT *
                             FROM   deleted
                             WHERE  deleted.AccountId =
                                                 AccountContact.AccountId)
              GROUP BY AccountId
              HAVING SUM(CASE WHEN PrimaryContactFlag = 1 then 1 ELSE 0 END) <> 1)

          THROW  50000, 'Account(s) do not have only one primary contact.', 1;
   END TRY
   BEGIN CATCH
        IF XACT_STATE() <> 0
            ROLLBACK TRANSACTION;
        THROW;
     END CATCH
END;
```

```
SELECT AccountId
FROM   Examples.AccountContact
--correlates the changed rows in inserted to the other rows
--for the account, so we can check if the rows have changed
-------------------------------------------------------------
WHERE  EXISTS (SELECT *
               FROM   inserted
               WHERE  inserted.AccountId = AccountContact.AccountId
               UNION ALL
               SELECT *
               FROM   deleted
               WHERE  deleted.AccountId = AccountContact.AccountId)
-------------------------------------------------------------
GROUP BY AccountId
HAVING SUM(CASE WHEN PrimaryContactFlag = 1 then 1 ELSE 0 END;
```

```
SELECT @AccountId = AccountId FROM inserted;
```

```
--Success, 1 row
INSERT INTO Examples.AccountContact(AccountContactId, AccountId, PrimaryContactFlag)
VALUES (1,1,1);
--Success, two rows
INSERT INTO Examples.AccountContact(AccountContactId, AccountId, PrimaryContactFlag)
VALUES (2,2,1),(3,3,1);
--Two rows, same account
INSERT INTO Examples.AccountContact(AccountContactId, AccountId, PrimaryContactFlag)
VALUES (4,4,1),(5,4,0);
--Invalid, two accounts with primary
INSERT INTO Examples.AccountContact(AccountContactId, AccountId, PrimaryContactFlag)
VALUES (6,5,1),(7,5,1);
```

```
Msg 50000, Level 16, State 1, Procedure AccountContact_TriggerAfterInsert, Line 29
One or more Accounts does not have one and only one primary contact.
```

```
--Invalid, no primary
INSERT INTO Examples.AccountContact(AccountContactId, AccountId, PrimaryContactFlag)
VALUES (8,6,0),(9,6,0);
--Won't work, because AccountId is new, and this row is not primary
UPDATE Examples.AccountContact
SET    AccountId = 6
WHERE  AccountContactId = 5;
```

```
CREATE TRIGGER Examples.AccountContact_TriggerAfterDelete
ON Examples.AccountContact
AFTER DELETE AS
BEGIN
   SET NOCOUNT ON;
   SET ROWCOUNT 0; --in case the client has modified the rowcount
   BEGIN TRY
   IF EXISTS ( SELECT AccountId
               FROM   Examples.AccountContact
               WHERE  EXISTS (SELECT *
                              FROM   deleted
                              WHERE  deleted.AccountId =
                                         AccountContact.AccountId)
            GROUP BY AccountId
            HAVING SUM(CASE WHEN PrimaryContactFlag = 1 then 1 ELSE 0 END) > 1)
       THROW  50000, 'One or more Accounts did not have one primary contact.', 1;
   END TRY
   BEGIN CATCH
       IF XACT_STATE() <> 0
           ROLLBACK TRANSACTION;
       THROW;
    END CATCH;
END;
```

```
CREATE TABLE Examples.Promise
(
    PromiseId int NOT NULL CONSTRAINT PKPromise PRIMARY KEY,
    PromiseAmount money NOT NULL
);
```

```
CREATE TABLE Examples.VerifyPromise
(
    VerifyPromiseId int NOT NULL CONSTRAINT PKVerifyPromise PRIMARY KEY,
    PromiseId int NOT NULL CONSTRAINT AKVerifyPromise UNIQUE
                --FK not included for simplicity
);
```

```
CREATE TRIGGER Examples.Promise_TriggerInsertUpdate
ON Examples.Promise
AFTER INSERT, UPDATE AS
BEGIN
   SET NOCOUNT ON;
   SET ROWCOUNT 0; --in case the client has modified the rowcount
   BEGIN TRY
       INSERT INTO Examples.VerifyPromise(PromiseId)
       SELECT PromiseId
       FROM   inserted
       WHERE  PromiseAmount > 10000.00
         AND  NOT EXISTS (SELECT * --keep from inserting duplicates
                          FROM   VerifyPromise
                          WHERE  VerifyPromise.PromiseId = inserted.PromiseId)
   END TRY
   BEGIN CATCH
       IF XACT_STATE() <> 0
           ROLLBACK TRANSACTION;
       THROW; --will halt the batch or be caught by the caller's catch block
   END CATCH
END;
```

```
CREATE TABLE Examples.Lamp
(
    LampId       int IDENTITY(1,1) CONSTRAINT PKLamp PRIMARY KEY,
    Value          varchar(10) NOT NULL,
    RowCreatedTime datetime2(0) NOT NULL
        CONSTRAINT DFLTLamp_RowCreatedTime DEFAULT(SYSDATETIME()),
    RowLastModifiedTime datetime2(0) NOT NULL
        CONSTRAINT DFLTLamp_RowLastModifiedTime DEFAULT(SYSDATETIME())
);
```

```
CREATE TRIGGER Examples.Lamp_TriggerInsteadOfInsert
ON Examples.Lamp
INSTEAD OF INSERT AS
BEGIN
   SET NOCOUNT ON;
   SET ROWCOUNT 0; --in case the client has modified the rowcount
   BEGIN TRY
        --skip columns to automatically set
        INSERT INTO Examples.Lamp( Value)
        SELECT Value
        FROM   inserted
   END TRY
   BEGIN CATCH
       IF XACT_STATE() <> 0
           ROLLBACK TRANSACTION;
       THROW; --will halt the batch or be caught by the caller's catch block
   END CATCH
END;
```

```
INSERT INTO Examples.Lamp(Value, RowCreatedTime, RowLastModifiedTime)
VALUES ('Original','1900-01-01','1900-01-01');

SELECT *
FROM   Examples.Lamp;
```

```
LampId      Value      RowCreatedTime              RowLastModifiedTime
----------- ---------- --------------------------- ---------------------------
1           Original   2016-09-20 21:03:54         2016-09-20 21:03:54
```

```
CREATE TRIGGER Examples.Lamp_TriggerInsteadOfUpdate
ON Examples.Lamp
INSTEAD OF UPDATE AS
BEGIN
   SET NOCOUNT ON;
   SET ROWCOUNT 0; --in case the client has modified the rowcount
   BEGIN TRY
        UPDATE Lamp
        SET    Value = inserted.Value,
               RowLastModifiedTime = DEFAULT --use default constraint
        FROM   Examples.Lamp
                 JOIN inserted
                    ON Lamp.LampId = inserted.LampId;
   END TRY
   BEGIN CATCH
       IF XACT_STATE() <> 0
           ROLLBACK TRANSACTION;
       THROW; --will halt the batch or be caught by the caller's catch block
   END CATCH;
END;
```

```
UPDATE Examples.Lamp
SET    Value = 'Modified',
       RowCreatedTime = '1900-01-01',
       RowLastModifiedTime = '1900-01-01'
WHERE LampId = 1;

SELECT *
FROM   Examples.Lamp;
```

```
LampId      Value      RowCreatedTime              RowLastModifiedTime
----------- ---------- --------------------------- ---------------------------
1           Modified   2016-09-20 21:07:07         2016-09-20 21:10:26
```

```
CREATE TABLE Examples.KeyTable1
(
    KeyValue int NOT NULL CONSTRAINT PKKeyTable1 PRIMARY KEY,
    Value1   varchar(10) NULL
);
CREATE TABLE Examples.KeyTable2
(
    KeyValue int NOT NULL CONSTRAINT PKKeyTable2 PRIMARY KEY,
    Value2    varchar(10) NULL
);
GO
CREATE VIEW Examples.KeyTable
AS
    SELECT COALESCE(KeyTable1.KeyValue, KeyTable2.KeyValue) as KeyValue,
           KeyTable1.Value1, KeyTable2.Value2
    FROM   Examples.KeyTable1
             FULL OUTER JOIN Examples.KeyTable2
                ON KeyTable1.KeyValue = KeyTable2.KeyValue;
```

```
INSERT INTO Examples.KeyTable (KeyValue, Value1, Value2)
VALUES (1,'Value1','Value2');
```

```
Msg 4406, Level 16, State 1, Line 21
Update or insert of view or function 'Examples.KeyTable' failed because it contains
a derived or constant field.
```

```
CREATE TRIGGER Examples.KeyTable_InsteadOfInsertTrigger
ON Examples.KeyTable
INSTEAD OF INSERT
AS
BEGIN
    SET NOCOUNT ON;
    SET ROWCOUNT 0; --in case the client has modified the rowcount
    BEGIN TRY
          --Insert data into one of the tables
        INSERT INTO Examples.KeyTable1(KeyValue, Value1)
        SELECT KeyValue, Value1
        FROM   Inserted;
        --and then the other
        INSERT INTO Examples.KeyTable2(KeyValue, Value2)
        SELECT KeyValue, Value2
        FROM   Inserted;
    END TRY
    BEGIN CATCH
        IF XACT_STATE() <> 0
            ROLLBACK TRANSACTION;
        THROW; --will halt the batch or be caught by the caller's catch block
    END CATCH;
END;
```

```
INSERT INTO Examples.KeyTable (KeyValue, Value1, Value2)
VALUES (1,'Value1','Value2');
```

```
KeyValue    Value1      Value2
----------- ----------- ----------
1           Value1      Value2
```

```
USE ExamBook762Ch2;
GO
CREATE TABLE Examples.DDLDatabaseChangeLog
(
   DDLDatabaseChangeLogId int NOT NULL IDENTITY
        CONSTRAINT PKDDLDatabaseChangeLog PRIMARY KEY,
    LogTime datetime2(0) NOT NULL,
    DDLStatement nvarchar(max) NOT NULL,
    LoginName sysname NOT NULL
);
```

```
--Names used to make it clear where you have used examples from this book outside
--of primary database
CREATE LOGIN Exam762Examples_DDLTriggerLogging WITH PASSWORD = 'PASSWORD$1';
CREATE USER Exam762Examples_DDLTriggerLogging
                                    FOR LOGIN Exam762Examples_DDLTriggerLogging;
GRANT INSERT ON  Examples.DDLDatabaseChangeLog TO
                               Exam762Examples_DDLTriggerLogging;
```

```
CREATE TRIGGER DatabaseCreations_ServerDDLTrigger
ON ALL SERVER
WITH EXECUTE AS 'Exam762Examples_DDLTriggerLogging'
FOR CREATE_DATABASE, ALTER_DATABASE, DROP_DATABASE
AS
    SET NOCOUNT ON;
    --trigger is stored in master db, so must
    INSERT INTO ExamBook762Ch2.Examples.DDLDatabaseChangeLog(LogTime, DDLStatement,
                                                                    LoginName)
    SELECT SYSDATETIME(),EVENTDATA().value(
                    '(/EVENT_INSTANCE/TSQLCommand/CommandText)[1]','nvarchar(max)'),
           ORIGINAL_LOGIN(); --Original login gives you the user that is connected.
                             --Otherwise we would get the EXECUTE AS user.
```

```
CREATE LOGIN Exam762Examples_DatabaseCreator WITH PASSWORD = 'PASSWORD$1';
GRANT CREATE ANY DATABASE TO Exam762Examples_DatabaseCreator;
GRANT ALTER ANY DATABASE TO Exam762Examples_DatabaseCreator;
```

```
CREATE DATABASE Example
GO
ALTER DATABASE Example SET RECOVERY SIMPLE;
GO
DROP DATABASE Example;
```

```
SELECT LogTime, DDLStatement, LoginName
FROM Examples.DDLDatabaseChangeLog;
```

```
LogTime                     DDLStatement                                    LoginName
--------------------------- ----------------------------------------------- -------------
2016-09-21 16:55:09         CREATE DATABASE Example                         Exam762Examp…
2016-09-21 16:55:19         ALTER DATABASE Example SET RECOVERY SIMPLE      Exam762Examp…
2016-09-21 16:55:27         DROP DATABASE Example                           Exam762Examp…
```

```
CREATE TRIGGER DatabaseCreations_StopThemAll
ON ALL SERVER
FOR CREATE_DATABASE, ALTER_DATABASE, DROP_DATABASE
AS
    SET NOCOUNT ON;
    ROLLBACK TRANSACTION;
    THROW 50000,'No more databases created please',1;
```

```
DISABLE TRIGGER DatabaseCreations_StopThemAll ON ALL SERVER;
```

```
DROP TRIGGER DatabaseCreations_ServerDDLTrigger ON ALL SERVER;
DROP USER Exam762Examples_DDLTriggerLogging;
DROP LOGIN Exam762Examples_DDLTriggerLogging;
DROP LOGIN Exam762Examples_DatabaseCreator;
```

```
CREATE TABLE Examples.DDLChangeLog
(
   DDLChangeLogId int NOT NULL IDENTITY
        CONSTRAINT PKDDLChangeLog PRIMARY KEY,
    LogTime datetime2(0) NOT NULL,
    DDLStatement nvarchar(max) NOT NULL,
    LoginName sysname NOT NULL
);
```

```
CREATE USER Exam762Examples_DDLTriggerLogging WITHOUT LOGIN;
GRANT INSERT ON Examples.DDLChangeLog TO Exam762Examples_DDLTriggerLogging;
```

```
CREATE TRIGGER DatabaseChanges_DDLTrigger
ON DATABASE
WITH EXECUTE AS 'Exam762Examples_DDLTriggerLogging'
FOR CREATE_TABLE, ALTER_TABLE, DROP_TABLE
AS
    SET NOCOUNT ON;
    DECLARE @eventdata XML = EVENTDATA();
    ROLLBACK; --Make sure the event doesn't occur
    INSERT INTO Examples.DDLChangeLog(LogTime, DDLStatement, LoginName)
    SELECT SYSDATETIME(),
           @EventData.value('(/EVENT_INSTANCE/TSQLCommand/CommandText)[1]',
                                                                'nvarchar(max)'),
           ORIGINAL_LOGIN();
    THROW 50000,'Denied!',1;
```

```
CREATE TABLE Examples.Test
(
    TestId int NOT NULL
);
GO
DROP TABLE Examples.DDLChangeLog;
```

```
Msg 50000, Level 16, State 1, Procedure DatabaseChanges_DDLTrigger, Line 25
Denied!
```

```
SELECT LogTime, DDLStatement, LoginName
FROM   Examples.DDLChangeLog;
```

```
LogTime                     DDLStatement                          LoginName
--------------------------- ------------------------------------- -----------------------
2016-09-21 19:16:06         CREATE TABLE Examples.Test            DomainName\louis
                            (
                                 TestId int NOT NULL
                            )
2016-09-21 19:16:51         DROP TABLE Examples.DDLChangeLog;     DomainName\louis
```

```
DROP TRIGGER DatabaseChanges_DDLTrigger ON DATABASE;
DROP USER Exam762Examples_DDLTriggerLogging;
```

```
CREATE TABLE Examples.LoginLog
(
    LoginLogId  int NOT NULL IDENTITY(1,1)
          CONSTRAINT PKLoginLog PRIMARY KEY,
    LoginName   sysname NOT NULL,
    LoginTime   datetime2(0) NOT NULL ,
    ApplicationName sysname NOT NULL
);
```

```
CREATE LOGIN Exam762Examples_LogonTriggerLogging WITH PASSWORD = 'PASSWORD$1';
CREATE USER Exam762Examples_LogonTriggerLogging
                                    FOR LOGIN Exam762Examples_LogonTriggerLogging;
GRANT INSERT ON Examples.LoginLog TO Exam762Examples_LogonTriggerLogging;
```

```
CREATE TRIGGER Exam762ExampleLogonTrigger
ON ALL SERVER
WITH EXECUTE AS 'Exam762Examples_LogonTriggerLogging'
FOR LOGON
AS
    IF ORIGINAL_LOGIN() = 'Login_NotAllowed'
        THROW 50000,'Unauthorized Access',1;
    ELSE
        INSERT INTO ExamBook762Ch2.Examples.LoginLog(LoginName, LoginTime,
                                                     ApplicationName)
        VALUES (ORIGINAL_LOGIN(),SYSDATETIME(),APP_NAME());
```

```
CREATE LOGIN Login_NotAllowed WITH PASS

WORD = 'PASSWORD$1';
```

```
   LoginName                LoginTime            ApplicationName
------------------------ -------------------- ----------------------------------------
WIN-8F59BO5AP7D\louis    2016-09-21 21:26:50  Microsoft SQL Se..IntelliSense
WIN-8F59BO5AP7D\louis    2016-09-21 21:26:50  Microsoft SQL Se..IntelliSense
WIN-8F59BO5AP7D\louis    2016-09-21 21:27:06  Microsoft SQL Se..Query
WIN-8F59BO5AP7D\louis    2016-09-21 21:27:06  Microsoft SQL Se..Query
```

```
DROP TRIGGER Exam762ExampleLogonTrigger ON ALL SERVER;
DROP USER Exam762Examples_LogonTriggerLogging;
DROP LOGIN Exam762Examples_LogonTriggerLogging;
```

```
CREATE TABLE Examples.UpdateRows
(
    UpdateRowsId int NOT NULL IDENTITY(1,1)
        CONSTRAINT PKUpdateRows PRIMARY KEY,
    Value varchar(20) NOT NULL
);
INSERT INTO Examples.UpdateRows (Value)
VALUES ('Original'),('Original'),('Original');
```

```
UpdateRowsId Value
------------ --------------------
1            Original
2            Original
3            Original
```

```
CREATE TRIGGER Examples.UpdateRows_TriggerInsert
ON Examples.UpdateRows
AFTER UPDATE AS
BEGIN
   SET NOCOUNT ON;
   SET ROWCOUNT 0;
   BEGIN TRY
        DECLARE @UpdateRowsId int
        SELECT @UpdateRowsId = UpdateRowsId
        FROM   inserted
        ORDER BY UpdateRowsId;

        UPDATE Examples.UpdateRows
        SET    Value = UPPER(Value)
        WHERE  UpdateRowsId = @UpdateRowsId;
   END TRY
   BEGIN CATCH
       IF XACT_STATE() <> 0
           ROLLBACK TRANSACTION;
       THROW; --will halt the batch or be caught by the caller's catch block
   END CATCH;

END;
```

```
UPDATE Examples.UpdateRows
SET    Value = 'Modified';
```

```
UpdateRowsId Value
------------ --------------------
1            Modified
2            Modified
3            MODIFIED

UpdateRowsId Value
------------ --------------------
1            Original
2            Original
3            Original
UpdateRowsId Value
------------ --------------------
1            MODIFIED
2            MODIFIED
3            MODIFIED

UpdateRowsId Value
------------ --------------------
1            MODIFIED
2            Modified
3            Modified
```

```
SELECT @UpdateRowsId = UpdateRowsId
FROM   inserted
ORDER BY UpdateRowsId;
```

```
CREATE TABLE Examples.KeyModify
(
    KeyModifyId  int CONSTRAINT PKKeyModify PRIMARY KEY,
    Value        varchar(20)
);
INSERT INTO Examples.KeyModify(KeyModifyId, Value)
VALUES (1,'Original'), (2,'Original'),(3,'Original');
```

```
CREATE TRIGGER Examples.KeyModify_TriggerInsteadOfInsert
ON Examples.KeyModify
INSTEAD OF UPDATE AS
BEGIN
   SET NOCOUNT ON;
   SET ROWCOUNT 0;
   BEGIN TRY
        UPDATE Examples.KeyModify
        SET    Value = UPPER(inserted.Value)
        FROM   Examples.KeyModify
                 JOIN inserted
                    ON KeyModify.KeyModifyId = inserted.KeyModifyId
   END TRY
   BEGIN CATCH
       IF XACT_STATE() <> 0
           ROLLBACK TRANSACTION;
       THROW;
   END CATCH
END;
```

```
UPDATE Examples.KeyModify
SET     KeyModifyId = KeyModifyId + 10, --Change Primary Key Value
        Value = 'Modified';
```

```
KeyModifyId Value
----------- --------------------
11          Modified
12          Modified
13          Modified
```

```
KeyModifyId Value
----------- --------------------
1           Original
2           Original
3           Original
```

```
KeyModifyId Value
----------- --------------------
1           Original
2           Original
3           Original
```

```
CREATE FUNCTION Examples.ReturnIntValue
(
    @Value  int
)
RETURNS int
AS
  BEGIN
    RETURN @Value
  END;
```

```
SELECT Functions.ReturnIntValue(1) as IntValue;
```

```
SELECT OrderId, 1/ (4732-OrderId)
FROM   Sales.Orders;
```

```
CREATE FUNCTION Sales.Customers_ReturnOrderCount
(
    @CustomerID int,
    @OrderDate date = NULL
)
RETURNS INT
WITH RETURNS NULL ON NULL INPUT, --if all parameters NULL, return NULL immediately
     SCHEMABINDING --make certain that the tables/columns referenced cannot change
AS
  BEGIN
      DECLARE @OutputValue int

      SELECT  @OutputValue = COUNT(*)
      FROM    Sales.Orders
      WHERE   CustomerID = @CustomerID
        AND   (OrderDate = @OrderDate
               OR @OrderDate IS NULL);

      RETURN @OutputValue
   END;
```

```
SELECT Sales.Customers_ReturnOrderCount(905, '2013-01-01');
```

```
SELECT Sales.Customers_ReturnOrderCount(905, DEFAULT);
```

```
SELECT CustomerID, Sales.Customers_ReturnOrderCount(905, DEFAULT)
FROM   Sales.Customers;

SELECT CustomerID, COUNT(*)
FROM   Sales.Orders
GROUP  BY CustomerID;
```

```
Table 'Worktable'. Scan count 0, logical reads 0, physical reads
Table 'Customers'. Scan count 1, logical reads 4, physical reads
 SQL Server Execution Times:
    CPU time = 375 ms,  elapsed time = 439 ms.
```

```
Table 'Orders'. Scan count 1, logical reads 191, physical reads 0
SQL Server Execution Times:
   CPU time = 16 ms,  elapsed time = 34 ms.
```

```
SELECT N'CPO' + RIGHT(N'00000000' + CustomerPurchaseOrderNumber,8)
FROM WideWorldImporters.Sales.Orders;
```

```
CREATE FUNCTION Sales.Orders_ReturnFormattedCPO
(
    @CustomerPurchaseOrderNumber nvarchar(20)
)
RETURNS nvarchar(20)
WITH RETURNS NULL ON NULL INPUT,
     SCHEMABINDING
AS
 BEGIN
    RETURN (N'CPO' + RIGHT(N'00000000' + @CustomerPurchaseOrderNumber,8));
 END;
```

```
SELECT Sales.Orders_ReturnFormattedCPO('12345') as CustomerPurchaseOrderNumber;
```

```
SELECT OrderId
FROM   Sales.Orders
WHERE  Sales.Orders_ReturnFormattedCPO(CustomerPurchaseOrderNumber) = 'CPO00019998';
```

```
SET STATISTICS TIME ON;
SELECT Sales.Orders_ReturnFormattedCPO(CustomerPurchaseOrderNumber)
FROM   Sales.Orders;

SELECT N'CPO' + RIGHT(N'00000000' + [CustomerPurchaseOrderNumber],8)
FROM WideWorldImporters.Sales.Orders;
```

```
CREATE FUNCTION Sales.Customers_ReturnOrderCountSetSimple
(
    @CustomerID int,
    @OrderDate date = NULL
)
RETURNS TABLE
AS
RETURN (SELECT COUNT(*) AS SalesCount,
               CASE WHEN MAX(BackorderOrderId) IS NOT NULL
                         THEN 1 ElSE 0 END AS HasBackorderFlag
        FROM   Sales.Orders
        WHERE  CustomerID = @CustomerID
        AND   (OrderDate = @OrderDate
               OR @OrderDate IS NULL));
```

```
SELECT *
FROM   Sales.Customers_ReturnOrderCountSetSimple(905,'2013-01-01');
```

```
SalesCount  HasBackorderFlag
----------  ----------------
2           1
```

```
SELECT *
FROM   Sales.Customers_ReturnOrderCountSetSimple(905,DEFAULT);
```

```
SalesCount  HasBackorderFlag
----------  ----------------
125         1
```

```
SELECT CustomerId, FirstDaySales.SalesCount, FirstDaySales.HasBackorderFlag
FROM   Sales.Customers
        OUTER APPLY Sales.Customers_ReturnOrderCountSetSimple
                              (CustomerId, AcountOpenedDate) as FirstDaySales
WHERE  FirstDaySales.SalesCount > 0;
```

CustomerId	SalesCount	HasBackorderFlag
10	2	1
57	1	0
...	...	...
995	2	1
1000	2	1

```
CREATE FUNCTION Sales.Customers_ReturnOrderCountSetMulti
(
    @CustomerID int,
    @OrderDate date = NULL
)
RETURNS  @OutputValue TABLE (SalesCount int NOT NULL,
                             HasBackorderFlag bit NOT NULL)
AS
 BEGIN
    INSERT INTO @OutputValue (SalesCount, HasBackorderFlag)
    SELECT COUNT(*) as SalesCount,
                  CASE WHEN MAX(BackorderOrderId) IS NOT NULL
                                THEN 1 ElSE 0 END AS HasBackorderFlag
    FROM   Sales.Orders
    WHERE  CustomerID = @CustomerID
    AND   (OrderDate = @OrderDate
            OR @OrderDate IS NULL)

    RETURN;
END;
```

```
SET STATISTICS TIME ON;
SELECT CustomerId, FirstDaySales.SalesCount, FirstDaySales.HasBackorderFlag
FROM   Sales.Customers
        OUTER APPLY Sales.Customers_ReturnOrderCountSetSimple
                        (CustomerId, AccountOpenedDate) as FirstDaySales
WHERE  FirstDaySales.SalesCount > 0;

SELECT CustomerId, FirstDaySales.SalesCount, FirstDaySales.HasBackorderFlag
FROM   Sales.Customers
        OUTER APPLY Sales.Customers_ReturnOrderCountSetMulti
                        (CustomerId, AccountOpenedDate) as FirstDaySales
WHERE  FirstDaySales.SalesCount > 0;
```

```
CREATE FUNCTION Examples.UpperCaseFirstLetter
(
    @Value varchar(50)
)
RETURNS nvarchar(50)
WITH SCHEMABINDING
AS
BEGIN
   --start at position 2, as 1 will always be uppercase if it exists
   DECLARE @OutputValue nvarchar(50), @position int = 2, @previousPosition int
   IF LEN(@Value) = 0 RETURN @OutputValue;
                      --remove leading spaces, uppercase the first character
   SET @OutputValue = (LTRIM(CONCAT(UPPER(SUBSTRING(@Value,1,1)),
                                          LOWER(SUBSTRING(@Value,2,99)))));
   --if no space characters, exit
   IF CHARINDEX(' ',@OutputValue,1) = 0 RETURN @OutputValue;
   WHILE 1=1
    BEGIN
       SET @position = CHARINDEX(' ',@outputValue,@position) + 1
       IF @position < @previousPosition or @position = 0
           BREAK;
       SELECT @OutputValue = CONCAT(SUBSTRING(@OutputValue,1,@position - 1),
                                    UPPER(SUBSTRING(@OutputValue,@position,1)),
                                    SUBSTRING(@OutputValue,@position + 1,50)),
              @PreviousPosition = @Position
    END
    RETURN @OutputValue
END;
```

```
SELECT Examples.UpperCaseOnlyFirstLetter(N'NO MORE YELLING') as Name;
```

```
Name
-----------------------------------------------
No More Yelling
```

```
SELECT OBJECTPROPERTY(OBJECT_ID('Examples.UpperCaseFirstLetter'), 'IsDeterministic')
                                                                    AS IsDeterministic
```

```
CREATE FUNCTION Examples.StartOfCurrentMonth
()
RETURNS date
WITH SCHEMABINDING
AS
 BEGIN
    RETURN (DATEADD(day, 0, DATEDIFF(day, 0, SYSDATETIME() ) -
                                          DATEPART(DAY,SYSDATETIME()) + 1));
 END;
```

```
SELECT OBJECTPROPERTY(OBJECT_ID('Examples.StartOfCurrentMonth'), 'IsDeterministic')
                                                                  AS IsDeterministic
```

```
CREATE FUNCTION Examples.ReturnOneToTenSet
()
RETURNS @OutputTable TABLE (I int)
WITH SCHEMABINDING
AS
  BEGIN
    INSERT INTO @OutputTable(I)
    VALUES (1),(2),(3),(4),(5),(6),(7),(8),(9),(10);

    RETURN;
  END;
```

```
SELECT OBJECTPROPERTY(OBJECT_ID('Examples.ReturnOneToTenSet'), 'IsDeterministic')
                                                                    AS IsDeterministic;
```

```
CREATE TABLE Examples.Respondent
(

     RespondentId int NOT NULL CONSTRAINT PKRespondent PRIMARY KEY,
     EmailAddress  nvarchar(500) NOT NULL
);
```

```
ALTER TABLE Examples.Respondent
     ADD CONSTRAINT AKRespondent UNIQUE (EmailAddress);
```

```
CREATE TABLE Examples.ThreeInsert
(
	ThreeInsertId int CONSTRAINT PKThreeInsert PRIMARY KEY
);
```

```
CREATE PROCEDURE Examples.ThreeInsert_Create
            @SecondValue int = 2 --Pass in 1 to and no data is inserted

AS
    SET NOCOUNT ON;
    BEGIN TRY
        BEGIN TRANSACTION;
        INSERT INTO Examples.ThreeInsert (ThreeInsertId)
        VALUES (1);
       INSERT INTO Examples.ThreeInsert (ThreeInsertId)
       VALUES (@SecondValue);
       COMMIT TRANSACTION;
    END TRY
    BEGIN CATCH
        IF XACT_STATE() <> 0
            ROLLBACK TRANSACTION;
        --No THROW will mean no reporting of message
    END CATCH;

    INSERT INTO Examples.ThreeInsert (ThreeInsertId)
    VALUES (3);
```

```
CREATE TABLE Examples.Recipient
(
    RecipientType varchar(30) NOT NULL
);
```

```
ALTER TABLE Examples.Recipient
   ADD CONSTRAINT CHKRecipient_RecipientType
         CHECK (RecipientType IN ('Regular','Special Handling'));
```

```
CREATE TABLE Examples.RecipientType
(
       RecipientType varchar(30) NOT NULL CONSTRAINT PKRecipientType PRIMARY KEY
);

INSERT INTO Examples.RecipientType(RecipientType)
VALUES ('Regular'),('Special Handling');

ALTER TABLE Examples.Recipient
      ADD CONSTRAINT FKRecipient_Ref_ExamplesRecipientType
      FOREIGN KEY (RecipientType) REFERENCES Examples.
RecipientType(RecipientType);
```

```
CREATE TABLE Examples.Offer
(
    OfferCode char(5) NOT NULL
);
```

```
CREATE TRIGGER Examples.Offer_TriggerInsteadOfInsert
ON Examples.Offer
INSTEAD OF INSERT AS
BEGIN
   SET NOCOUNT ON;
   SET ROWCOUNT 0; --in case the client has modified the rowcount
   BEGIN TRY
        IF EXISTS (SELECT *
                   FROM   inserted
                   WHERE  OfferCode NOT LIKE '[A-Z][A-Z][A-Z][A-Z][A-Z]')
            THROW 50000,'An OfferCode is not all alpha characters',1;

        --skip columns to automatically set
        INSERT INTO Examples.Offer (OfferCode)
        SELECT UPPER(OfferCode)
        FROM   inserted
   END TRY
   BEGIN CATCH
       IF XACT_STATE() <> 0
           ROLLBACK TRANSACTION;
       THROW; --will halt the batch or be caught by the caller's catch block
   END CATCH
END;
```

```
SELECT collation_name
FROM sys.databases
WHERE  database_id = DB_ID();
```

```
ALTER TABLE Examples.Offer
    ADD CONSTRAINT CHKOffer_OfferCode
       CHECK (OfferCode LIKE '[A-Z][A-Z][A-Z][A-Z][A-Z]'
                             COLLATE Latin1_General_100_CS_AS);
```

```
CREATE TRIGGER DatabaseChanges_DDLTrigger
ON DATABASE
WITH EXECUTE AS 'Exam762Examples_DDLTriggerLogging'
FOR CREATE_INDEX
AS
    SET NOCOUNT ON;
    IF CAST(SYSDATETIME() AS time) >= '08:00:00'
        AND CAST(SYSDATETIME() AS time) < '10:00:00'

    THROW 50000,'No indexes may be added between 8 and 10 AM',1;
```

```
IF @checkAll IS NULL
      THROW 50000,'The value of @checkAll may not be NULL',1;
```

```
CREATE DATABASE ExamBook762Ch3;
GO
USE ExamBook762Ch3;
GO
CREATE SCHEMA Examples;
GO
CREATE TABLE Examples.TestParent
(
    ParentId  int NOT NULL
        CONSTRAINT PKTestParent PRIMARY KEY,
    ParentName  varchar(100) NULL
);

CREATE TABLE Examples.TestChild
(
    ChildId  int NOT NULL
        CONSTRAINT PKTestChild PRIMARY KEY,
    ParentId int NOT NULL,
    ChildName  varchar(100) NULL
);

ALTER TABLE Examples.TestChild
    ADD CONSTRAINT FKTestChild_Ref_TestParent
        FOREIGN KEY (ParentId) REFERENCES Examples.TestParent(ParentId);

INSERT INTO Examples.TestParent(ParentId, ParentName)
VALUES (1, 'Dean'),(2, 'Michael'),(3, 'Robert');

INSERT INTO Examples.TestChild (ChildId, ParentId, ChildName)
VALUES (1,1, 'Daniel'), (2, 1, 'Alex'), (3, 2, 'Matthew'), (4, 3, 'Jason');
```

```
SELECT ParentId, ParentName
FROM Examples.TestParent;
```

```
BEGIN TRANSACTION;
    UPDATE Examples.TestParent
    SET ParentName = 'Mike'
    WHERE ParentName = 'Michael';

    UPDATE Examples.TestChild
    SET ChildName = 'Matt'
    WHERE ChildName = 'Matthew';
COMMIT TRANSACTION;
```

```
SELECT TestParent.ParentId, ParentName, ChildId, ChildName
FROM Examples.TestParent
    FULL OUTER JOIN Examples.TestChild ON TestParent.ParentId = TestChild.ParentId;
```

ParentId	ParentName	ChildId	ChildName
1	Dean	1	Daniel
1	Dean	2	Alex
2	Michael	3	Matt
3	Bob	4	Jason

```
      BEGIN TRANSACTION;
          INSERT INTO Examples.TestParent(ParentId, ParentName)
          VALUES (4, 'Linda');

    DELETE Examples.TestParent
    WHERE ParentName = 'Bob';
COMMIT TRANSACTION;
```

```
(1 row(s) affected)
Msg 547, Level 16, State 0, Line 24
The DELETE statement conflicted with the REFERENCE constraint "FKTestChild_Ref_
TestParent". The conflict occurred in database "ExamBook762Ch3", table "Examples.
TestChild", column 'ParentId'.

The statement has been terminated.
```

```
    SET XACT_ABORT ON;
    BEGIN TRANSACTION;
        INSERT INTO Examples.TestParent(ParentId, ParentName)
        VALUES (5, 'Isabelle');

    DELETE Examples.TestParent
    WHERE ParentName = 'Bob';
COMMIT TRANSACTION;
```

```
        SET XACT_ABORT OFF;
        BEGIN TRANSACTION;
            INSERT INTO Examples.TestParent(ParentId, ParentName)
            VALUES (5, 'Isabelle');

    DELETE Examples.TestParent
    WHEN ParentName = 'Bob';
COMMIT TRANSACTION;
```

```
BEGIN TRY
    BEGIN TRANSACTION;
        INSERT INTO Examples.TestParent(ParentId, ParentName)
        VALUES (5, 'Isabelle');

        DELETE Examples.TestParent
        WHERE ParentName = 'Bob';
    COMMIT TRANSACTION;
END TRY
BEGIN CATCH
    IF @@TRANCOUNT > 0 ROLLBACK TRANSACTION;
END CATCH
```

```
BEGIN TRANSACTION;
    INSERT INTO Examples.TestParent(ParentId, ParentName)
    VALUES (5, 'Isabelle');
```

```
SET IMPLICIT_TRANSACTIONS ON;
```

```
INSERT INTO Examples.TestParent(ParentId, ParentName)
VALUES (6, 'Lukas');
SELECT @@TRANCOUNT;
```

```
SET IMPLICIT_TRANSACTIONS OFF;
```

```
BEGIN TRANSACTION;
    INSERT INTO Examples.TestParent(ParentId, ParentName)
    VALUES (7, 'Mary');
    DELETE Examples.TestParent
    WHERE ParentName = 'Bob';
IF @@ERROR != 0
    BEGIN
        ROLLBACK TRANSACTION;
    RETURN
END
COMMIT TRANSACTION;
```

```
CREATE PROCEDURE Examples.DeleteParent
    @ParentId INT
AS
    BEGIN TRANSACTION;
        DELETE Examples.TestParent
        WHERE ParentId = @ParentId;
    IF @@ERROR != 0
        BEGIN
            ROLLBACK TRANSACTION;
            RETURN;
        END
    COMMIT TRANSACTION;
GO
BEGIN TRANSACTION;
    INSERT INTO Examples.TestParent(ParentId, ParentName)
        VALUES (7, 'Mary');
        EXEC Examples.DeleteParent @ParentId=3;
    IF @@ERROR != 0
        BEGIN
            ROLLBACK TRANSACTION;
        RETURN
    END
    COMMIT TRANSACTION;
    GO
```

```
(1 row(s) affected)
Msg 547, Level 16, State 0, Procedure DeleteParent, Line 6 [Batch Start Line 16]
The DELETE statement conflicted with the REFERENCE constraint
"FKTestChild_Ref_TestParent". The conflict occurred in database "ExamBook762Ch3", table
"Examples.TestChild", column 'ParentId'.
The statement has been terminated.
Msg 266, Level 16, State 2, Procedure DeleteParent, Line 0 [Batch Start Line 16]
Transaction count after EXECUTE indicates a mismatching number of BEGIN and COMMIT
statements. Previous count = 1, current count = 0.
Msg 3903, Level 16, State 1, Line 25
The ROLLBACK TRANSACTION request has no corresponding BEGIN TRANSACTION.
```

```
CREATE PROCEDURE Examples.DeleteParentNoNest
    @ParentId INT
AS
    DECLARE @CurrentTranCount INT;
    SELECT @CurrentTranCount = @@TRANCOUNT;
    IF (@CurrentTranCount = 0)
        BEGIN TRANSACTION DeleteTran;
    ELSE
        SAVE TRANSACTION DeleteTran;
    DELETE Examples.TestParent
    WHERE ParentId = @ParentId;
    IF @@ERROR != 0
        BEGIN
            ROLLBACK TRANSACTION DeleteTran;
            RETURN;
        END
    IF (@CurrentTranCount = 0)
        COMMIT TRANSACTION;
GO
BEGIN TRANSACTION;
    INSERT INTO Examples.TestParent(ParentId, ParentName)
    VALUES (7, 'Mary');
    EXEC Examples.DeleteParentNoNest @ParentId=3;
IF @@ERROR != 0
    BEGIN
        ROLLBACK TRANSACTION;
    RETURN
END
COMMIT TRANSACTION;
GO
```

```
BEGIN TRANSACTION;
    INSERT INTO Examples.TestParent(ParentId, ParentName)
    VALUES (8, 'Ed');
    SAVE TRANSACTION StartTran;

    SELECT 'StartTran' AS Status, ParentId, ParentName
    FROM Examples.TestParent;

    DELETE Examples.TestParent
        WHERE ParentId = 7;
    SAVE TRANSACTION DeleteTran;

    SELECT 'Delete 1' AS Status, ParentId, ParentName
    FROM Examples.TestParent;

    DELETE Examples.TestParent
        WHERE ParentId = 6;
    SELECT 'Delete 2' AS Status, ParentId, ParentName
    FROM Examples.TestParent;

    ROLLBACK TRANSACTION DeleteTran;
    SELECT 'RollbackDelete2' AS Status, ParentId, ParentName
    FROM Examples.TestParent;

    ROLLBACK TRANSACTION StartTran;
    SELECT @@TRANCOUNT AS 'TranCount';
    SELECT 'RollbackStart' AS Status, ParentId, ParentName
    FROM Examples.TestParent;
COMMIT TRANSACTION;
GO
```

```
Status    ParentId    ParentName
--------- ----------- ------------------------------------------------------------------
StartTran 1           Dean
StartTran 2           Mike
StartTran 3           Bob
StartTran 4           Linda
StartTran 5           Isabelle
StartTran 6           Lukas
StartTran 7           Mary
StartTran 8           Ed

Status   ParentId    ParentName
-------- ----------- ------------------------------------------------------------------
Delete 1 1           Dean
Delete 1 2           Mike
Delete 1 3           Bob
Delete 1 4           Linda
Delete 1 5           Isabelle
Delete 1 6           Lukas
Delete 1 8           Ed

Status   ParentId    ParentName
-------- ----------- ------------------------------------------------------------------
Delete 2 1           Dean
Delete 2 2           Mike
Delete 2 3           Bob
```

```
Delete 2 4           Linda
Delete 2 5           Isabelle
Delete 2 8           Ed

Status          ParentId    ParentName
--------------- ----------- ------------------------------------------------------------
RollbackDelete2 1           Dean
RollbackDelete2 2           Mike
RollbackDelete2 3           Bob
RollbackDelete2 4           Linda
RollbackDelete2 5           Isabelle
RollbackDelete2 6           Lukas
RollbackDelete2 8           Ed

TranCount
-----------
1

Status        ParentId    ParentName
------------- ----------- ------------------------------------------------------------
RollbackStart 1           Dean
RollbackStart 2           Mike
RollbackStart 3           Bob
RollbackStart 4           Linda
RollbackStart 5           Isabelle
RollbackStart 6           Lukas
RollbackStart 7           Mary
RollbackStart 8           Ed
```

```
BEGIN TRANSACTION;
SELECT ParentId, ParentName
FROM Examples.TestParent WITH (HOLDLOCK);
WAITFOR DELAY '00:00:15';
ROLLBACK TRANSACTION;
```

```
INSERT INTO Examples.TestParent WITH (TABLOCK)
SELECT <columns> FROM <table>;
```

```
CREATE TABLE Examples.IsolationLevels
(
    RowId  int NOT NULL
        CONSTRAINT PKRowId PRIMARY KEY,
    ColumnText  varchar(100) NOT NULL
);

INSERT INTO Examples.IsolationLevels(RowId, ColumnText)
VALUES (1, 'Row 1'), (2, 'Row 2'), (3, 'Row 3'), (4, 'Row 4');
```

```
SET TRANSACTION ISOLATION LEVEL READ COMMITTED;
```

```
BEGIN TRANSACTION;
    UPDATE Examples.IsolationLevels
        SET ColumnText = 'Row 1 Updated'
        WHERE RowId = 1;
```

```
SELECT RowId, ColumnText
FROM Examples.IsolationLevels;
```

```
BEGIN TRANSACTION;
    UPDATE Examples.IsolationLevels
        SET ColumnText = 'Row 1 Updated'
        WHERE RowId = 1;
```

```
SET TRANSACTION ISOLATION LEVEL READ UNCOMMITTED;
SELECT RowId, ColumnText
FROM Examples.IsolationLevels;
```

```
SELECT RowId, ColumnText
FROM Examples.IsolationLevels;
```

```
SELECT RowId, ColumnText
FROM Examples.IsolationLevels
WITH (NOLOCK);
```

```
SET TRANSACTION ISOLATION LEVEL REPEATABLE READ;
BEGIN TRANSACTION;
    SELECT RowId, ColumnText
    FROM Examples.IsolationLevels;
    WAITFOR DELAY '00:00:15';
    SELECT RowId, ColumnText
    FROM Examples.IsolationLevels;
ROLLBACK TRANSACTION;
```

```
UPDATE Examples.IsolationLevels
    SET ColumnText = 'Row 1 Updated'
    WHERE RowId = 1;
```

```
SET TRANSACTION ISOLATION LEVEL REPEATABLE READ;
BEGIN TRANSACTION;
    SELECT RowId, ColumnText
    FROM Examples.IsolationLevels;
    WAITFOR DELAY '00:00:15';
    SELECT RowId, ColumnText
    FROM Examples.IsolationLevels;
ROLLBACK TRANSACTION;
```

```
INSERT INTO Examples.IsolationLevels(RowId, ColumnText)
VALUES (5, 'Row 5');
```

```
SET TRANSACTION ISOLATION LEVEL SERIALIZABLE;
```

```
INSERT INTO Examples.IsolationLevels(RowId, ColumnText)
VALUES (6, 'Row 6');
```

```
ALTER DATABASE ExamBook762Ch3
SET ALLOW_SNAPSHOT_ISOLATION ON;
```

```
SET TRANSACTION ISOLATION LEVEL SNAPSHOT;
BEGIN TRANSACTION;
    SELECT RowId, ColumnText
    FROM Examples.IsolationLevels;
    WAITFOR DELAY '00:00:15';
    SELECT RowId, ColumnText
    FROM Examples.IsolationLevels;
ROLLBACK TRANSACTION;
```

```
INSERT INTO Examples.IsolationLevels(RowId, ColumnText)
VALUES (7, 'Row 7');
```

```
CREATE DATABASE ExamBook762Ch3_IsolationTest;
GO
USE ExamBook762Ch3_IsolationTest;
GO
CREATE SCHEMA Examples;
GO
CREATE TABLE Examples.IsolationLevelsTest
(RowId INT NOT NULL
    CONSTRAINT PKRowId PRIMARY KEY,
    ColumnText  varchar(100) NOT NULL
);
INSERT INTO Examples.IsolationLevelsTest(RowId, ColumnText)
VALUES (1, 'Row 1'), (2, 'Row 2'), (3, 'Row 3'), (4, 'Row 4');
```

```
SET TRANSACTION ISOLATION LEVEL SNAPSHOT;
BEGIN TRANSACTION;
    SELECT t1.RowId, t2.ColumnText
    FROM Examples.IsolationLevels AS t1
    INNER JOIN ExamBook762Ch3_IsolationTest.Examples.IsolationLevelsTest AS t2
    ON t1.RowId = t2.RowId;
END TRANSACTION;
```

```
Msg 3952, Level 16, State 1, Line 5
Snapshot isolation transaction failed accessing database 'ExamBook762Ch3_IsolationTest'
because snapshot isolation is not allowed in this database. Use ALTER DATABASE to allow
snapshot isolation.
```

```
SET TRANSACTION ISOLATION LEVEL SNAPSHOT;
BEGIN TRANSACTION;
    SELECT t1.RowId, t2.ColumnText
    FROM Examples.IsolationLevels AS t1
    INNER JOIN ExamBook762Ch3_IsolationTest.Examples.IsolationLevelsTest AS t2
    WITH (READCOMMITTED)
    ON t1.RowId = t2.RowId;
END TRANSACTION;
```

```
CREATE INDEX Ix_RowId ON Examples.IsolationLevels (RowId);
```

```
SET TRANSACTION ISOLATION LEVEL SNAPSHOT;
BEGIN TRANSACTION;
    SELECT RowId, ColumnText
    FROM Examples.IsolationLevels;
    WAITFOR DELAY '00:00:15';
    SELECT RowId, ColumnText
    FROM Examples.IsolationLevels;
ROLLBACK TRANSACTION;
```

```
ALTER INDEX Ix_RowId
    ON Examples.IsolationLevels REBUILD;
```

```
Msg 3961, Level 16, State 1, Line 6
Snapshot isolation transaction failed in database 'ExamBook762Ch3' because the object
accessed by the statement has been modified by a DDL statement in another concurrent
transaction since the start of this transaction.  It is disallowed because the metadata
is not versioned. A concurrent update to metadata can lead to inconsistency if mixed
with snapshot isolation.
```

```
ALTER DATABASE ExamBook762Ch3
SET ALLOW_SNAPSHOT_ISOLATION OFF;
```

```
ALTER DATABASE ExamBook762Ch3
SET READ_COMMITTED_SNAPSHOT ON;
```

```
BEGIN TRANSACTION;
    SELECT RowId, ColumnText
    FROM Examples.IsolationLevels;
    WAITFOR DELAY '00:00:15';
    SELECT RowId, ColumnText
    FROM Examples.IsolationLevels;
ROLLBACK TRANSACTION;
```

```
INSERT INTO Examples.IsolationLevels(RowId, ColumnText)
VALUES (8, 'Row 8');
```

```
ALTER DATABASE Examples
SET READ_COMMITTED_SNAPSHOT OFF;
```

```
CREATE TABLE Examples.LockingA
(
    RowId  int NOT NULL
        CONSTRAINT PKLockingARowId PRIMARY KEY,
    ColumnText  varchar(100) NOT NULL
);

INSERT INTO Examples.LockingA(RowId, ColumnText)
VALUES (1, 'Row 1'), (2, 'Row 2'), (3, 'Row 3'), (4, 'Row 4');
CREATE TABLE Examples.LockingB
(
    RowId  int NOT NULL
        CONSTRAINT PKLockingBRowId PRIMARY KEY,
    ColumnText  varchar(100) NOT NULL
);

INSERT INTO Examples.LockingB(RowId, ColumnText)
VALUES (1, 'Row 1'), (2, 'Row 2'), (3, 'Row 3'), (4, 'Row 4');
```

```
BEGIN TRANSACTION;
    SELECT RowId, ColumnText
    FROM Examples.LockingA
    WITH (HOLDLOCK, ROWLOCK);
```

```
BEGIN TRANSACTION;
    UPDATE Examples.LockingA
        SET ColumnText = 'Row 2 Updated'
        WHERE RowId = 2;
```

```
SELECT
    request_session_id as s_id,
    resource_type,
    resource_associated_entity_id,
    request_status,
    request_mode
FROM sys.dm_tran_locks
WHERE resource_database_id = db_id('ExamBook762Ch3');
```

s_id	resource_type	resource_associated_entity_id	request_status	request_mode
1	DATABASE	0	GRANT	S
2	DATABASE	0	GRANT	S
1	PAGE	72057594041729024	GRANT	IS
2	PAGE	72057594041729024	GRANT	IX
1	KEY	72057594041729024	GRANT	RangeS-S
1	KEY	72057594041729024	GRANT	RangeS-S
1	KEY	72057594041729024	GRANT	RangeS-S
1	KEY	72057594041729024	GRANT	RangeS-S
1	KEY	72057594041729024	GRANT	RangeS-S
2	KEY	72057594041729024	WAIT	X
1	OBJECT	933578364	GRANT	IS
2	OBJECT	933578364	GRANT	IX

```
SELECT
    object_name(object_id) as Resource,
    object_id,
    hobt_id
FROM sys.partitions
WHERE hobt_id=72057594041729024;
```

```
Resource object_id  hobt_id
-------- ---------- -------------------
LockingA 933578364  72057594041729024
```

```
SELECT
    t1.resource_type AS res_typ,
    t1.resource_database_id AS res_dbid,
    t1.resource_associated_entity_id AS res_entid,
    t1.request_mode AS mode,
    t1.request_session_id AS s_id,
    t2.blocking_session_id AS blocking_s_id
FROM sys.dm_tran_locks as t1
INNER JOIN sys.dm_os_waiting_tasks as t2
    ON t1.lock_owner_address = t2.resource_address;
```

```
res_typ  res_dbid   res_entid           mode  s_id   blocking_s_id
-------  --------   ------------------  ----  ------  -----------------
KEY      27         720575940417290 24  X     2      1
```

```
SELECT
    wait_type as wait,
    waiting_tasks_count as wt_cnt,
    wait_time_ms as wt_ms,
    max_wait_time_ms as max_wt_ms,
    signal_wait_time_ms as signal_ms
FROM sys.dm_os_wait_stats
WHERE wait_type LIKE 'LCK%'
ORDER BY wait_time_ms DESC;
```

wait	wt_cnt	wt_ms	max_wt_ms	signal_ms
LCK_M_X	6	1170670	712261	114
LCK_M_S	28	19398	2034	43
LCK_M_SCH_M	449	92	28	46
LCK_M_SCH_S	1	72	72	0

```
SELECT
    wait_type as wait,
    wait_time_ms as wt_ms,
    CONVERT(decimal(9,2), 100.0 * wait_time_ms /
    SUM(wait_time_ms) OVER ()) as wait_pct
FROM sys.dm_os_wait_stats
WHERE wait_type LIKE 'LCK%'
ORDER BY wait_time_ms DESC;
```

```
BEGIN TRANSACTION;
    UPDATE Examples.LockingA
        SET ColumnText = 'Row 1 Updated'
        WHERE RowId = 1;
    WAITFOR DELAY '00:00:05';
    UPDATE Examples.LockingB;
    SET ColumnText = 'Row 1 Updated Again'
    WHERE RowId = 1;
```

```
BEGIN TRANSACTION;
    UPDATE Examples.LockingB
        SET ColumnText = 'Row 1 Updated'
        WHERE RowId = 1;
    WAITFOR DELAY '00:00:05';
    UPDATE Examples.LockingA;
    SET ColumnText = 'Row 1 Updated Again'
    WHERE RowId = 1;
```

```
Msg 1205, Level 13, State 51, Line 6
Transaction (Process ID 70) was deadlocked on lock resources with another process and
has been chosen as the deadlock victim. Rerun the transaction.
```

```
BEGIN TRANSACTION;
    UPDATE Examples.LockingA
        SET ColumnText = 'Row 2 Updated'
        WHERE RowId = 2;
    WAITFOR DELAY '00:00:05';
    UPDATE Examples.LockingB
    SET ColumnText = 'Row 2 Updated Again'
    WHERE RowId = 2;
```

```
BEGIN TRANSACTION;
    UPDATE Examples.LockingB
        SET ColumnText = 'Row 2 Updated'
        WHERE RowId = 2;
    WAITFOR DELAY '00:00:05';
    UPDATE Examples.LockingA
    SET ColumnText = 'Row 2 Updated Again'
    WHERE RowId = 2;
```

```
DECLARE @Tries tinyint
SET @Tries = 1
WHILE @Tries <= 3
BEGIN

    BEGIN TRANSACTION
    BEGIN TRY
        UPDATE Examples.LockingB
            SET ColumnText = 'Row 3 Updated'
            WHERE RowId = 3;
        WAITFOR DELAY '00:00:05';
        UPDATE Examples.LockingA
        SET ColumnText = 'Row 3 Updated Again'
            WHERE RowId = 3;
        COMMIT TRANSACTION;
    END TRY
    BEGIN CATCH
        SELECT ERROR_NUMBER() AS ErrorNumber;
        ROLLBACK TRANSACTION;
        SET @Tries = @Tries + 1;
        CONTINUE;
    END CATCH
END
```

```
CREATE DATABASE ExamBook762Ch3_IMOLTP
ON PRIMARY (
    NAME = ExamBook762Ch3_IMOLTP_data,
    FILENAME = 'C:\data\ExamBook762Ch3_IMOLTP.mdf', size=500MB
),
FILEGROUP ExamBook762Ch3_IMOLTP_FG CONTAINS MEMORY_OPTIMIZED_DATA (
    NAME = ExamBook762Ch3_IMOLTP_FG_Container,
    FILENAME = 'C:\data\ExamBook762Ch3_IMOLTP_FG_Container'
)
LOG ON (
    NAME = ExamBook762Ch3_IMOLTP_log,
    FILENAME = 'C:\data\ExamBook762Ch3_IMOLTP_log.ldf', size=500MB
);
GO
```

```
USE ExamBook762Ch3_IMOLTP;
GO
CREATE SCHEMA Examples;
GO
CREATE TABLE Examples.Order_Disk (
    OrderId INT NOT NULL PRIMARY KEY NONCLUSTERED,
    OrderDate DATETIME NOT NULL,
    CustomerCode NVARCHAR(5) NOT NULL
);
GO
CREATE TABLE Examples.Order_IM (
    OrderID INT NOT NULL PRIMARY KEY NONCLUSTERED,
    OrderDate DATETIME NOT NULL,
    CustomerCode NVARCHAR(5) NOT NULL
)
WITH (MEMORY_OPTIMIZED = ON);
GO
```

```
USE ExamBook762Ch3_IMOLTP;
GO
-- Create natively compiled stored procedure
CREATE PROCEDURE Examples.OrderInsert_NC
    @OrderID INT,
    @CustomerCode NVARCHAR(10)
WITH NATIVE_COMPILATION, SCHEMABINDING
AS
BEGIN ATOMIC
WITH (TRANSACTION ISOLATION LEVEL = SNAPSHOT, LANGUAGE = N'English')
    DECLARE @OrderDate DATETIME = getdate();
    INSERT INTO Examples.Order_IM (OrderId, OrderDate, CustomerCode)
    VALUES (@OrderID, @OrderDate, @CustomerCode);
END;
GO
-- Create interpreted stored procedure
CREATE PROCEDURE Examples.OrderInsert_Interpreted
    @OrderID INT,
    @CustomerCode NVARCHAR(10),
    @TargetTable NVARCHAR(20)
AS
    DECLARE @OrderDate DATETIME = getdate();
    DECLARE @SQLQuery NVARCHAR(MAX);
    SET @SQLQuery = 'INSERT INTO ' +
        @TargetTable +
        ' (OrderId, OrderDate, CustomerCode) VALUES (' +
        CAST(@OrderID AS NVARCHAR(6)) +
        ','''' +  CONVERT(NVARCHAR(20), @OrderDate, 101)+
        ''','''' +  @CustomerCode +
        ''')';
    EXEC (@SQLQuery);
GO
```

```
SET STATISTICS TIME OFF;
SET NOCOUNT ON;

DECLARE @starttime DATETIME = sysdatetime();
DECLARE @timems INT;
DECLARE @i INT = 1;
DECLARE @rowcount INT = 100000;
DECLARE @CustomerCode NVARCHAR(10);

--Reset disk-based table
TRUNCATE TABLE Examples.Order_Disk;

-- Disk-based table and interpreted stored procedure
BEGIN TRAN;
    WHILE @i <= @rowcount
    BEGIN;
        SET @CustomerCode = 'cust' + CAST(@i as NVARCHAR(6));
        EXEC Examples.OrderInsert_Interpreted @i, @CustomerCode, 'Examples.Order_Disk';
        SET @i += 1;
    END;
COMMIT;

SET @timems = datediff(ms, @starttime, sysdatetime());
SELECT 'Disk-based table and interpreted stored procedure: ' AS [Description],
    CAST(@timems AS NVARCHAR(10)) + ' ms' AS Duration;
-- Memory-based table and interpreted stored procedure
SET @i = 1;
SET @starttime = sysdatetime();
```

```
BEGIN TRAN;
    WHILE @i <= @rowcount
    BEGIN;
        SET @CustomerCode = 'cust' + CAST(@i AS NVARCHAR(6));
        EXEC Examples.OrderInsert_Interpreted @i, @CustomerCode, 'Examples.Order_IM';
        SET @i += 1;
    END;
COMMIT;

SET @timems = datediff(ms, @starttime, sysdatetime());
SELECT 'Memory-optimized table and interpreted stored procedure: ' AS [Description],
    CAST(@timems AS NVARCHAR(10)) + ' ms' AS Duration;

-- Reset memory-optimized table
DELETE FROM Examples.Order_IM;
SET @i = 1;
SET @starttime = sysdatetime();

BEGIN TRAN;
    WHILE @i <= @rowcount
    BEGIN;
        SET @CustomerCode = 'cust' + CAST(@i AS NVARCHAR(6));
        EXEC Examples.OrderInsert_NC @i, @CustomerCode;
        SET @i += 1;
    END;
COMMIT;

SET @timems = datediff(ms, @starttime, sysdatetime());
SELECT 'Memory-optimized table and natively compiled stored procedure:'
    AS [Description],
    CAST(@timems AS NVARCHAR(10)) + ' ms' AS Duration;
GO
```

```
Description                                          Duration
---------------------------------------------------- ---------
Disk-based table and interpreted stored procedure:   10440 ms

Description                                                Duration
---------------------------------------------------------- ---------
Memory-optimized table and interpreted stored procedure:   10041 ms

Description                                                          Duration
-------------------------------------------------------------------- ---------
Memory-optimized table and natively compiled stored procedure:       1885 ms
```

```
CREATE TABLE Examples.Order_IM_Hash (
    OrderID INT NOT NULL PRIMARY KEY
        NONCLUSTERED HASH WITH (BUCKET_COUNT = 1000000),
    OrderDate DATETIME NOT NULL,
    CustomerCode NVARCHAR(5) NOT NULL
        INDEX ix_CustomerCode HASH WITH (BUCKET_COUNT = 1000000)
)
WITH (MEMORY_OPTIMIZED = ON);
```

```
CREATE TABLE Examples.Order_IM_CCI (
    OrderID INT NOT NULL PRIMARY KEY NONCLUSTERED,
    OrderDate DATETIME NOT NULL,
    CustomerCode NVARCHAR(5) NOT NULL,
    INDEX ix_CustomerCode_cci CLUSTERED COLUMNSTORE)
WITH (MEMORY_OPTIMIZED = ON);
```

```
CREATE TABLE Examples.Order_IM_NC (
    OrderID INT NOT NULL PRIMARY KEY NONCLUSTERED,
    OrderDate DATETIME NOT NULL,
    CustomerCode NVARCHAR(5) NOT NULL INDEX ix_CustomerCode NONCLUSTERED
)
WITH (MEMORY_OPTIMIZED = ON);
```

```
USE ExamBook762Ch3_IMOLTP;
GO
-- Add a column and an index
ALTER TABLE Examples.Order_IM
    ADD Quantity INT NULL,
    INDEX ix_OrderDate(OrderDate);
-- Alter an index by changing the bucket count
ALTER TABLE Examples.Order_IM_Hash
    ALTER INDEX ix_CustomerCode
        REBUILD WITH ( BUCKET_COUNT = 2000000);
-- Drop an index
ALTER TABLE Examples.Order_IM
    DROP INDEX ix_OrderDate;
```

```
CREATE TABLE Examples.Order_IM_Durable (
    OrderID INT NOT NULL PRIMARY KEY NONCLUSTERED,
    OrderDate DATETIME NOT NULL,
    CustomerCode NVARCHAR(5) NOT NULL
)
WITH (MEMORY_OPTIMIZED = ON, DURABILITY=SCHEMA_AND_DATA);
GO
```

```
CREATE TABLE Examples.Order_IM_Nondurable (
    OrderID INT NOT NULL PRIMARY KEY NONCLUSTERED,
    OrderDate DATETIME NOT NULL,
    CustomerCode NVARCHAR(5) NOT NULL
)
WITH (MEMORY_OPTIMIZED = ON, DURABILITY=SCHEMA_ONLY);
GO
```

```
--Set at database level only, all transactions commit as delayed durable
ALTER DATABASE ExamBook762Ch3_IMOLTP
    SET DELAYED_DURABILITY = FORCED;
--Override database delayed durability at commit for durable transaction
BEGIN TRANSACTION;
    INSERT INTO Examples.Order_IM_Hash
    (OrderId, OrderDate, CustomerCode)
    VALUES (1, getdate(), 'cust1');
COMMIT TRANSACTION WITH (DELAYED_DURABILITY = OFF);
GO

--Set at transaction level only
ALTER DATABASE ExamBook762Ch3_IMOLTP
    SET DELAYED_DURABILITY = ALLOWED;
BEGIN TRANSACTION;
    INSERT INTO Examples.Order_IM_Hash
    (OrderId, OrderDate, CustomerCode)
    VALUES (2, getdate(), 'cust2');
COMMIT TRANSACTION WITH (DELAYED_DURABILITY = ON);

--Set within a natively compiled stored procedure
CREATE PROCEDURE Examples.OrderInsert_NC_DD
    @OrderID INT,
    @CustomerCode NVARCHAR(10)
WITH NATIVE_COMPILATION, SCHEMABINDING
AS
BEGIN ATOMIC
WITH (DELAYED_DURABILITY = ON,
        TRANSACTION ISOLATION LEVEL = SNAPSHOT, LANGUAGE = N'English')
    DECLARE @OrderDate DATETIME = getdate();
    INSERT INTO Examples.Order_IM (OrderId, OrderDate, CustomerCode)
    VALUES (@OrderID, @OrderDate, @CustomerCode);
END;
GO
--Disable delayed durability completely for all transactions
--    and natively compiled stored procedures
ALTER DATABASE ExamBook762Ch3_IMOLTP
    SET DELAYED_DURABILITY = DISABLED;
```

```
--Enable statistics collection at the procedure level
EXEC sys.sp_xtp_control_proc_exec_stats @new_collection_value = 1;

--Check the current status of procedure-level statistics collection
DECLARE @c BIT;
EXEC sys.sp_xtp_control_proc_exec_stats @old_collection_value=@c output
SELECT @c AS 'Current collection status';

--Disable statistics collection at the procedure level
EXEC sys.sp_xtp_control_proc_exec_stats @new_collection_value = 0;
```

```
--Enable statistics collection at the query level
EXEC sys.sp_xtp_control_query_exec_stats @new_collection_value = 1;

--Check the current status of query-level statistics collection
DECLARE @c BIT;
EXEC sys.sp_xtp_control_query_exec_stats @old_collection_value=@c output;
SELECT @c AS 'Current collection status';

--Disable statistics collection at the query level
EXEC sys.sp_xtp_control_query_exec_stats @new_collection_value = 0;

--Enable statistics collection at the query level for a specific
--natively compiled stored procedure
DECLARE @ncspid int;
DECLARE @dbid int;
SET @ncspid = OBJECT_ID(N'Examples.OrderInsert_NC');
SET @dbid = DB_ID(N'ExamBook762Ch3_IMOLTP')
EXEC [sys].[sp_xtp_control_query_exec_stats] @new_collection_value = 1,
    @database_id = @dbid, @xtp_object_id = @ncspid;

--Check the current status of query-level statistics collection for a specific
--natively compiled stored procedure
DECLARE @c bit;
DECLARE @ncspid int;
DECLARE @dbid int;
SET @ncspid = OBJECT_ID(N'Examples.OrderInsert_NC');
SET @dbid = DB_ID(N'ExamBook762Ch3_IMOLTP')
EXEC sp_xtp_control_query_exec_stats @database_id = @dbid,
    @xtp_object_id = @ncspid, @old_collection_value=@c output;
SELECT @c AS 'Current collection status';

--Disable statistics collection at the query level for a specific
--natively compiled stored procedure
DECLARE @ncspid int;
DECLARE @dbid int;
EXEC sys.sp_xtp_control_query_exec_stats @new_collection_value = 0,
    @database_id = @dbid, @xtp_object_id = @ncspid;
```

```
SELECT
    OBJECT_NAME(PS.object_id) AS obj_name,
    cached_time as cached_tm,
    last_execution_time as last_exec_tm,
    execution_count as ex_cnt,
    total_worker_time as wrkr_tm,
    total_elapsed_time as elpsd_tm
FROM sys.dm_exec_procedure_stats PS
INNER JOIN sys.all_sql_modules SM
    ON SM.object_id = PS.object_id
WHERE SM.uses_native_compilation = 1;
```

```
obj_name       cached_tm                last_exec_tm           ex_cnt wrkr_tm  elpsd_tm
---------      ----------------------- ----------------------- ------ -------- --------
OrderInsert_NC 2016-10-15 20:44:33.917 2016-10-15 20:44:35.273 100000 376987   383365
```

```
SELECT
    st.objectid as obj_id,
    OBJECT_NAME(st.objectid) AS obj_nm,
    SUBSTRING(st.text,
        (QS.statement_start_offset / 2 ) + 1,
        ((QS.statement_end_offset - QS.statement_start_offset) / 2) + 1)
            AS 'Query',
        QS.last_execution_time as last_exec_tm,
        QS.execution_count as ex_cnt
    FROM sys.dm_exec_query_stats QS
    CROSS APPLY sys.dm_exec_sql_text(sql_handle) st
    INNER JOIN sys.all_sql_modules SM
        ON SM.object_id = st.objectid
    WHERE SM.uses_native_compilation = 1
```

```
obj_id      obj_name     Query                  last_exec_tm            ex_cnt
-------   -------------- ---------------------- ----------------------- -------
981578535 OrderInsert_NC INSERT INTO            2016-10-15 21:09:25.877 100000
                          Examples.Order_IM
                          (OrderId, OrderDate,
                          CustomerCode)
                          VALUES (@OrderID,
                          @OrderDate,
                          @CustomerCode)
```

```
SELECT
     so.OrderID,
     OrderDate,
     ExpectedDeliveryDate,
     CustomerID,
     CustomerPurchaseOrderNumber,
     StockItemID,
     Quantity,
     UnitPrice
FROM Sales.Orders so
     WITH (NOLOCK)
INNER JOIN Sales.OrderLines sol
     WITH (NOLOCK)
     ON so.OrderID = sol.OrderID
```

```
BEGIN TRANSACTION;
    UPDATE <do something>;
    BEGIN TRANSACTION;
        UPDATE <DO SOMETHING>;
        BEGIN TRANSACTION;
            UPDATE <DO SOMETHING>;
COMMIT TRANSACTION;
```

```
SELECT CustomerName FROM Customer
WHERE StartDate > DateAdd(day, -7, GetUtcDate());
```

```
USE WideWorldImporters;
GO
CREATE NONCLUSTERED INDEX IX_Purchasing_Suppliers_ExamBook762Ch4
    ON Purchasing.Suppliers
(
    SupplierCategoryID,
    SupplierID
)
INCLUDE (SupplierName);
GO

DBCC SHOW_STATISTICS ('Purchasing.Suppliers',
    IX_Purchasing_Suppliers_ExamBook762Ch4 );

/* Output
Name           Updated       Rows      Rows Sampled      Steps   Density
Average key length   String Index     Filter Expression Unfiltered Rows
------------------------------ --------------------- ---- -------------
IX_Purchasing_Suppliers_ExamBook762Ch4 Nov  7 2016  6:40AM 13  13  5    1
8                  NO                          NULL                        13
(1 row(s) affected)

All density       Average Length Columns
-----------------  ------------ ----------------------------------------
0.125               4                                    SupplierCategoryID
0.07692308    8                              SupplierCategoryID, SupplierID
(2 row(s) affected)

RANGE_HI_KEY  RANGE_ROWS  EQ_ROWS    DISTINCT_RANGE_ROWS  AVG_RANGE_ROWS
-------------------- ------------------- --------------- --------------
2                  0                   6                  0                  1
4                  1                   1                  1                  1
6                  1                   1                  1                  1
8                  1                   1                  1                  1
9                  0                   1                  0                  1
(5 row(s) affected)
DBCC execution completed. If DBCC printed error messages, contact your system
administrator.
 */
```

```
CREATE DATABASE ExamBook762Ch4_Statistics;
GO
ALTER DATABASE ExamBook762Ch4_Statistics
    SET   AUTO_CREATE_STATISTICS OFF;
ALTER DATABASE ExamBook762Ch4_Statistics
    SET AUTO_UPDATE_STATISTICS OFF;
ALTER DATABASE ExamBook762Ch4_Statistics
    SET AUTO_UPDATE_STATISTICS_ASYNC OFF;
GO
USE ExamBook762Ch4_Statistics;
GO
CREATE SCHEMA Examples;
GO
CREATE TABLE Examples.OrderLines (
    OrderLineID int NOT NULL,
    OrderID int NOT NULL,
    StockItemID int NOT NULL,
    Description nvarchar(100) NOT NULL,
    PackageTypeID int NOT NULL,
    Quantity int NOT NULL,
    UnitPrice decimal(18, 2) NULL,
    TaxRate decimal(18, 3) NOT NULL,
    PickedQuantity int NOT NULL,
    PickingCompletedWhen datetime2(7) NULL,
    LastEditedBy int NOT NULL,
    LastEditedWhen datetime2(7) NOT NULL);
```

```
GO
INSERT INTO Examples.OrderLines
SELECT *
FROM WideWorldImporters.Sales.OrderLines;
GO
CREATE INDEX ix_OrderLines_StockItemID
    ON Examples.OrderLines (StockItemID);
GO
DBCC SHOW_STATISTICS ('Examples.OrderLines',
    ix_OrderLines_StockItemID );
GO

/* Partial Output
RANGE_HI_KEY  RANGE_ROWS  EQ_ROWS  DISTINCT_RANGE_ROWS  AVG_RANGE_ROWS
----------------- --------------- ----------- ----------------------- ---
1                 0                1048                 0                 1
2                 0                1078                 0                 1
4               1022               1066                 1              1022
*/
```

```
UPDATE Examples.OrderLines
    SET StockItemID = 1
    WHERE OrderLineID < 45000;
DBCC SHOW_STATISTICS ('Examples.OrderLines',
    ix_OrderLines_StockItemID );
GO
/* Partial Output
RANGE_HI_KEY  RANGE_ROWS  EQ_ROWS  DISTINCT_RANGE_ROWS  AVG_RANGE_ROWS
-------------- ------------- -------- --------------- ----------------
1                 0                1048               0               1
2                 0                1078               0               1
4              1022                1066               1            1022
*/
```

```
Use WideWorldImporters;
GO
SELECT
    OBJECT_NAME(object_id) AS ObjectName,
    name,
    auto_created
FROM sys.stats
WHERE auto_created = 1 AND
    object_id IN
        (SELECT object_id FROM sys.objects WHERE type = 'U');

/* Partial Output
ObjectName                    name                    auto_created
----------------------------------------- --------------------
Colors_Archive                 _WA_Sys_00000001_04E4BC85  1
OrderLines                     _WA_Sys_00000006_05A3D694  1
OrderLines                     _WA_Sys_0000000C_05A3D694  1
plan_persist_runtime_stats     _WA_Sys_00000006_0CBAE877  1
StockGroups_Archive           _WA_Sys_00000001_10566F31   1
StateProvinces                 _WA_Sys_00000002_114A936A  1
StateProvinces                 _WA_Sys_00000009_114A936A  1
CustomerTransactions           _WA_Sys_0000000B_15DA3E5D  1
*/
```

```
SELECT
    name AS ObjectName,
    STATS_DATE(object_id, stats_id) AS UpdateDate
FROM sys.stats
WHERE object_id = OBJECT_ID('Sales.Customers');

/* Output
ObjectName                                                      UpdateDate
------------------------------ ---------------------------------------
PK_Sales_Customers                                      2016-06-02 10:07:35.170
UQ_Sales_Customers_CustomerName                         2016-06-02 10:07:35.240
FK_Sales_Customers_CustomerCategoryID                   2016-06-02 10:08:13.080
FK_Sales_Customers_BuyingGroupID                        2016-06-02 10:07:38.010
FK_Sales_Customers_PrimaryContactPersonID               2016-06-02 10:07:43.027
FK_Sales_Customers_AlternateContactPersonID             2016-06-02 10:07:48.040
FK_Sales_Customers_DeliveryMethodID                     2016-06-02 10:07:53.043
FK_Sales_Customers_DeliveryCityID                       2016-06-02 10:07:58.060
FK_Sales_Customers_PostalCityID                         2016-06-02 10:08:03.060
IX_Sales_Customers_Perf_20160301_06                     2016-06-02 10:08:08.067
_WA_Sys_00000003_2FCF1A8A                               2016-06-02 10:10:11.130
_WA_Sys_0000000B_2FCF1A8A                               2016-06-02 10:10:11.623
_WA_Sys_0000001E_2FCF1A8A                               2016-06-02 10:54:31.173
*/
```

```
EXEC sp_configure 'show advanced options', 1;
GO
RECONFIGURE;
GO
EXEC sp_configure 'Agent XPs', 1;
GO
RECONFIGURE;
GO
```

```
USE WideWorldImporters;
GO
UPDATE STATISTICS [Application].[Cities]
WITH FULLSCAN
GO
```

```
SELECT
    OBJECT_NAME(ixu.object_id, DB_ID('WideWorldImporters')) AS [object_name] ,
    ix.[name] AS index_name ,
    ixu.user_seeks + ixu.user_scans + ixu.user_lookups AS user_reads,
    ixu.user_updates AS user_writes
FROM sys.dm_db_index_usage_stats ixu
INNER JOIN WideWorldImporters.sys.indexes ix ON
    ixu.[object_id] = ix.[object_id] AND
    ixu.index_id = ix.index_id
WHERE ixu.database_id = DB_ID('WideWorldImporters')
ORDER BY user_reads DESC;
/*Partial Output
object_name                index_name                   user_reads user_writes
------------------------------------------ -------------- ---------------
StockGroups                   PK_Warehouse_StockGroups    19                0
Suppliers                     PK_Purchasing_Suppliers     13                0
ColdRoomTemperatures_Archive  ix_ColdRoomTemperatures_Archive      0      1959
StockItems                    FK_Warehouse_StockItems_SupplierID   0         0
*/
```

```
USE WideWorldImporters;
GO
SELECT
    OBJECT_NAME(ix.object_id) AS ObjectName ,
    ix.name
FROM sys.indexes AS ix
INNER JOIN sys.objects AS o ON
    ix.object_id = o.object_id
WHERE ix.index_id NOT IN (
    SELECT ixu.index_id
    FROM sys.dm_db_index_usage_stats AS ixu
    WHERE
        ixu.object_id = ix.object_id AND
        ixu.index_id = ix.index_id AND
        database_id = DB_ID()
    ) AND
    o.[type] = 'U'
ORDER BY OBJECT_NAME(ix.object_id) ASC ;

/* Partial Output
ObjectName                          name
-------------------------------------------------------------------------------
BuyingGroups            UQ_Sales_BuyingGroups_BuyingGroupName
BuyingGroups_Archive    ix_BuyingGroups_Archive
Cities_Archive          ix_Cities_Archive
ColdRoomTemperatures    NULL
ColdRoomTemperatures    PK_Warehouse_ColdRoomTemperatures
ColdRoomTemperatures    IX_Warehouse_ColdRoomTemperatures_ColdRoomSensorNumber
*/
```

```
USE WideWorldImporters;
GO
SELECT
    o.name AS ObjectName ,
    ix.name AS IndexName ,
    ixu.user_seeks + ixu.user_scans + ixu.user_lookups AS user_reads ,
    ixu.user_updates AS user_writes ,
    SUM(p.rows) AS total_rows
FROM sys.dm_db_index_usage_stats ixu
INNER JOIN sys.indexes ix ON
    ixu.object_id = ix.object_id AND
    ixu.index_id = ix.index_id
INNER JOIN sys.partitions p ON
    ixu.object_id = p.object_id AND
    ixu.index_id = p.index_id
INNER JOIN sys.objects o ON
    ixu.object_id = o.object_id
WHERE
    ixu.database_id = DB_ID() AND
    OBJECTPROPERTY(ixu.object_id, 'IsUserTable') = 1 AND
    ixu.index_id > 0
GROUP BY
    o.name ,
    ix.name ,
    ixu.user_seeks + ixu.user_scans + ixu.user_lookups ,
    ixu.user_updates
HAVING ixu.user_seeks + ixu.user_scans + ixu.user_lookups = 0
ORDER BY
    ixu.user_updates DESC,
    o.name ,
    ix.name ;

/* Output
ObjectName                    IndexName    user_reads  user_writes  total_rows
------------------------- --------------- --------------- -----------------
ColdRoomTemperatures_Archive ix_ColdRoomTemperatures_Archive  0  2016  3654736
StockItems                    FK_Warehouse_StockItems_SupplierID 0   0      227
*/
```

```
DECLARE  @db_id SMALLINT, @object_id INT;
SET @db_id = DB_ID(N'WideWorldImporters');
SET @object_id = OBJECT_ID(N'WideWorldImporters.Sales.Orders');
SELECT
    ixs.index_id AS idx_id,
    ix.name AS ObjectName,
    index_type_desc,
    page_count,
    avg_page_space_used_in_percent AS AvgPageSpacePct,
    fragment_count AS frag_ct,
    avg_fragmentation_in_percent AS AvgFragPct
FROM sys.dm_db_index_physical_stats
    (@db_id, @object_id, NULL, NULL , 'Detailed') ixs
INNER JOIN sys.indexes ix ON
    ixs.index_id = ix.index_id AND
    ixs.object_id = ix.object_id
ORDER BY avg_fragmentation_in_percent DESC;

/* Output
idx_id ObjectName   index_type_desc    pg_ct AvgPageSpacePct   frag_ct  AvgFragPct
-------- ---------------------------------- ---------- --------------------------
4 FK_Sales_Orders_PickedByPersonID  NONCLUSTERED INDEX  237  53.6864838151717 237
                                                                  99.57805907173
2 FK_Sales_Orders_CustomerID        NONCLUSTERED INDEX   189 67.3274277242402 189
                                                                  97.8835978835979
5 FK_Sales_Orders_ContactPersonID   NONCLUSTERED INDEX  189  67.3274277242402 189
                                                                  97.8835978835979
3 FK_Sales_Orders_SalespersonPersonID NONCLUSTERED INDEX 136 93.5749444032617 136
                                                                  97.0588235294118
1 PK_Sales_Orders                    CLUSTERED INDEX    688 99.0945268099827   88
                                                                 0.290697674418605
1 PK_Sales_Orders                    CLUSTERED INDEX    2   55.2260934025204  2 0
1 PK_Sales_Orders                    CLUSTERED INDEX    1 0.296515937731653  1  0
2 FK_Sales_Orders_CustomerID         NONCLUSTERED INDEX  1 39.6713615023474 1   0
3 FK_Sales_Orders_SalespersonPersonID NONCLUSTERED INDEX  1 28.5396590066716 1  0
4 FK_Sales_Orders_PickedByPersonID    NONCLUSTERED INDEX  1 58.5001235483074 1  0
5 FK_Sales_Orders_ContactPersonID     NONCLUSTERED INDEX  1 39.6713615023474 1  0
*/
```

```
SELECT
    (user_seeks + user_scans) * avg_total_user_cost * (avg_user_impact * 0.01) AS
IndexImprovement,
    id.statement,
    id.equality_columns,
    id.inequality_columns,
    id.included_columns
FROM sys.dm_db_missing_index_group_stats AS igs
INNER JOIN sys.dm_db_missing_index_groups AS ig
    ON igs.group_handle = ig.index_group_handle
INNER JOIN sys.dm_db_missing_index_details AS id
    ON ig.index_handle = id.index_handle
ORDER BY IndexImprovement DESC;

/* Output
IndexImprovmeent  statement equality_columns   inequality_columns  included_columns
------------------------ ---------------------------- ------------------------
79.89008274829 [AdventureWorksDW].[dbo].[FactInternetSales] [ProductKey] NULL
                                                                        [CustomerKey]
*/
```

```
USE [WideWorldImporters];
GO
CREATE NONCLUSTERED INDEX [IX_Sales_Invoices_ExamBook762Ch4_A]
    ON [Sales].[Invoices]
(
    [CustomerID],
    [InvoiceDate]
)
INCLUDE ([TotalDryItems]);
GO
CREATE NONCLUSTERED INDEX [IX_Sales_Invoices_ExamBook762Ch4_B]
    ON [Sales].[Invoices]
(
    [CustomerID],
    [InvoiceDate],
    [CustomerPurchaseOrderNumber]
)
INCLUDE ([TotalDryItems]);
GO
```

```
USE [WideWorldImporters];
WITH IndexColumns AS (
    SELECT
        '[' + s.Name + '].[' + T.Name + ']' AS TableName,
        ix.name AS IndexName,
        c.name AS ColumnName,
        ix.index_id,
        ixc.index_column_id,
        COUNT(*) OVER(PARTITION BY t.OBJECT_ID, ix.index_id) AS ColumnCount
    FROM sys.schemas AS s
    INNER JOIN sys.tables AS t ON
        t.schema_id = s.schema_id
    INNER JOIN sys.indexes AS ix ON
        ix.OBJECT_ID = t.OBJECT_ID
    INNER JOIN sys.index_columns AS ixc ON
        ixc.OBJECT_ID = ix.OBJECT_ID AND
        ixc.index_id = ix.index_id
    INNER JOIN sys.columns AS c ON
        c.OBJECT_ID = ixc.OBJECT_ID AND
        c.column_id = ixc.column_id
WHERE
        ixc.is_included_column = 0 AND
        LEFT(ix.name, 2) NOT IN ('PK', 'UQ', 'FK')
```

```
)
SELECT DISTINCT
    ix1.TableName,
    ix1.IndexName AS Index1,
    ix2.IndexName AS Index2
FROM IndexColumns AS ix1
INNER JOIN IndexColumns AS ix2 ON
    ix1.TableName = ix2.TableName AND
    ix1.IndexName <> ix2.IndexName AND
    ix1.index_column_id = ix2.index_column_id AND
    ix1.ColumnName = ix2.ColumnName AND
    ix1.index_column_id < 3 AND
    ix1.index_id < ix2.index_id AND
    ix1.ColumnCount <= ix2.ColumnCount
ORDER BY ix1.TableName, ix2.IndexName;

/* Output
TableName        Index1                                              Index2
-------------------------------------------------------------------------
      [Sales].[Invoices] IX_Sales_Invoices_ExamBook762Ch4_A
                         IX_Sales_Invoices_ExamBook762Ch4_B
[Sales].[OrderLines]IX_Sales_OrderLines_AllocatedStockItems
                       IX_Sales_OrderLines_Perf_20160301_02
*/
```

```
IF EXISTS(SELECT *
    FROM sys.server_event_sessions
    WHERE name='ActualQueryPlans')
    DROP EVENT SESSION ActualQueryPlans
    ON SERVER;
GO
CREATE EVENT SESSION ActualQueryPlans
ON SERVER
ADD EVENT sqlserver.query_post_execution_showplan(
    ACTION (sqlserver.database_name,
                    sqlserver.client_hostname,
                    sqlserver.client_app_name,
                    sqlserver.plan_handle,
                    sqlserver.sql_text,
                    sqlserver.tsql_stack,
                    package0.callstack,
                    sqlserver.query_hash,
                    sqlserver.session_id,
                    sqlserver.request_id)
    WHERE
        sqlserver.database_name='WideWorldImporters' AND
        object_type = 'ADHOC'
)
ADD TARGET package0.event_file(SET filename=N'C:\ExamBook762Ch4\ActualQueryPlans.xel',
    max_file_size=(5),max_rollover_files=(4)),
ADD TARGET package0.ring_buffer
    WITH (MAX_DISPATCH_LATENCY=5SECONDS, TRACK_CAUSALITY=ON);
GO
ALTER EVENT SESSION ActualQueryPlans
    ON SERVER
    STATE=START;
GO
```

```
--Disable extended event session
ALTER EVENT SESSION ActualQueryPlans
    ON SERVER
    STATE=STOP;
GO
--Drop extended event session
IF EXISTS(SELECT *
    FROM sys.server_event_sessions
    WHERE name='ActualQueryPlans')
    DROP EVENT SESSION ActualQueryPlans
    ON SERVER;
GO
```

```
USE master;
GO
DECLARE @TraceID int;
EXEC sp_trace_create
    @TraceID output,
    0,
    N'C:\ExamBook762Ch4\ActualQueryPlanTrc';

EXEC sp_trace_setevent @TraceID,
    146,    -- Showplan XML Statistics Profile
    27,     -- BinaryData column
    1;      -- Column is ON for this event

EXEC sp_trace_setevent @TraceID,
    146,
    1,       -- TextData column
    1;

EXEC sp_trace_setevent @TraceID,
    146,
    14,      -- StartTime column
    1;

EXEC sp_trace_setevent @TraceID,
    146,
    15,      -- EndTime column
    1;
```

```
-- Set filter for database
EXEC sp_trace_setfilter @TraceID,
    @ColumnID = 35, --Database Name
    @LogicalOperator = 0, -- Logical AND
    @ComparisonOperator = 6, -- Comparison LIKE
    @Value = N'WideWorldImporters' ;

-- Set filter for application name
EXEC sp_trace_setfilter @TraceID,
    @ColumnID = 10, --ApplicationName
    @LogicalOperator = 0, -- Logical AND
    @ComparisonOperator = 6, -- Comparison LIKE
    @Value = N'Microsoft SQL Server Management Studio - Query' ;

-- Start Trace (status 1 = start)
EXEC sp_trace_setstatus @TraceID, 1;
GO
```

```
----  Find the trace ID
USE master;
GO
SELECT *
FROM sys.fn_trace_getinfo(0)
WHERE value = 'C:\ExamBook762Ch4\ActualQueryPlanTrc.trc';

-- Set  the trace status to stop
EXEC sp_trace_setstatus
    @traceid = <traceid>,
    @status= 0;
GO

-- Close and Delete the trace
EXEC sp_trace_setstatus
    @traceid = <traceid>,
    @status = 2;
GO
```

```
--Get event identifiers
SELECT
    e.trace_event_id AS EventID,
    e.name AS EventName,
    c.name AS CategoryName
FROM sys.trace_events e
JOIN sys.trace_categories c
    ON e.category_id = c.category_id
ORDER BY e.trace_event_id;

/* Partial output
EventID    EventName                      CategoryName
---------- ------------------------------ --------------------
10             RPC:Completed                   Stored Procedures
11             RPC:Starting                   Stored Procedures
12             SQL:BatchCompleted     TSQL
13             SQL:BatchStarting           TSQL
*/

--Get column identifiers for events
SELECT
    trace_column_id,
    name AS ColumnName
FROM sys.trace_columns

ORDER BY trace_column_id;

/* Partial output
trace_column_id  ColumnName
---------------------- ----------------------------
1                              TextData
2                              BinaryData
3                              DatabaseID
4                              TransactionID
5                              LineNumber
*/
```

```
CREATE VIEW Sales.vSalesByYear
WITH SCHEMABINDING
AS
    SELECT
    YEAR(InvoiceDate) AS InvoiceYear,
    COUNT_BIG(*) AS InvoiceCount
FROM Sales.Invoices
GROUP BY YEAR(InvoiceDate);
GO
CREATE UNIQUE CLUSTERED INDEX idx_vSalesByYear
    ON Sales.vSalesByYear
    (InvoiceYear);
GO
```

```
SELECT *
FROM Warehouse.VehicleTemperatures;
```

```
SELECT *
FROM Warehouse.StockGroups;
```

```
SELECT
    StockGroupID,
    StockGroupName,
    ValidFrom,
    ValidTo
FROM Warehouse.StockGroups
WHERE StockGroupName = 'Novelty Items';
```

```
OPTION(min_grant_percent = 100)
```

```
SELECT
    YEAR(InvoiceDate) AS InvoiceYear,
    COUNT(InvoiceID) AS InvoiceCount
FROM Sales.Invoices
GROUP BY YEAR(InvoiceDate);
```

```
DCREATE VIEW Sales.vSalesByYear
WITH SCHEMABINDING
AS
    SELECT
    YEAR(InvoiceDate) AS InvoiceYear,
    COUNT_BIG(*) AS InvoiceCount
FROM Sales.Invoices
GROUP BY YEAR(InvoiceDate);
GO
CREATE UNIQUE CLUSTERED INDEX idx_vSalesByYear
    ON Sales.vSalesByYear
    (InvoiceYear);
GO
```

```
SELECT
    YEAR(InvoiceDate) AS InvoiceYear,
    COUNT(InvoiceID) AS InvoiceCount
FROM Sales.Invoices
GROUP BY YEAR(InvoiceDate);
```

```
SELECT
     si.StockItemName,
     c.ColorName,
     s.SupplierName

FROM Warehouse.StockItems si
INNER JOIN Warehouse.Colors c ON
     c.ColorID = si.ColoriD
INNER JOIN Purchasing.Suppliers s ON
     s.SupplierID = si.SupplierID;
```

```
CREATE NONCLUSTERED INDEX IX_Purchasing_Suppliers_ExamBook762Ch4_SupplierID
    ON Purchasing.Suppliers
(
    SupplierID ASC,
    SupplierName
);
GO
CREATE NONCLUSTERED INDEX IX_Warehouse_StockItems_ExamBook762Ch4_ColorID
    ON Warehouse.StockItems
(
    ColorID ASC,
    SupplierID ASC,
    StockItemName ASC
);
```

```
    SELECT
      si.StockItemName,
      c.ColorName,
      s.SupplierName
FROM Warehouse.StockItems si
INNER JOIN Warehouse.Colors c ON
     c.ColorID = si.ColoriD
INNER JOIN Purchasing.Suppliers s ON
```

```
SELECT
     si.StockItemName,
     c.ColorName,
     s.SupplierName

FROM Warehouse.StockItems si
INNER JOIN Warehouse.Colors c ON
     c.ColorID = si.ColoriD
INNER JOIN Purchasing.Suppliers s ON
     s.SupplierID = si.SupplierID;
```

```
ALTER DATABASE <databasename>
    SET QUERY_STORE = ON
    (
        OPERATION_MODE = READ_WRITE ,
        CLEANUP_POLICY = ( STALE_QUERY_THRESHOLD_DAYS = 30 ),
        DATA_FLUSH_INTERVAL_SECONDS = 3000,
        MAX_STORAGE_SIZE_MB = 500,
        INTERVAL_LENGTH_MINUTES = 50
    );
```

```
--Option 1: Use the ALTER DATABASE statement
ALTER DATABASE <databasename>
SET QUERY_STORE CLEAR ALL;
GO

--Option 2: Use a system stored procedure
EXEC sys.sp_query_store_flush_db;
GO
```

```
USE WideWorldImporters;
GO
SELECT TOP 1
    qt.query_sql_text,
    CAST(query_plan AS XML) AS QueryPlan,
    rs.avg_logical_io_reads
FROM sys.query_store_plan qp
INNER JOIN sys.query_store_query q
  ON qp.query_id = q.query_id
INNER JOIN sys.query_store_query_text qt
    ON q.query_text_id = qt.query_text_id
INNER JOIN sys.query_store_runtime_stats rs
    ON qp.plan_id = rs.plan_id
ORDER BY rs.avg_logical_io_reads DESC;
```

```
CREATE DATABASE ExamBook762Ch4_QueryStore;
GO
USE ExamBook762Ch4_QueryStore;
GO
CREATE SCHEMA Examples;
GO
CREATE TABLE Examples.SimpleTable(
    Ident INT IDENTITY,
    ID INT,
    Value INT);
WITH IDs
    AS (SELECT
                TOP (9999)
                ROW_NUMBER() OVER (ORDER BY (SELECT 1)) AS n
            FROM master.sys.All_Columns ac1
            CROSS JOIN master.sys.All_Columns ac2
        )
INSERT  INTO Examples.SimpleTable(ID, Value)
SELECT
    1,
    n
FROM    IDs;
GO
INSERT Examples.SimpleTable (ID, Value)
    VALUES (2, 100);
ALTER TABLE Examples.SimpleTable
    ADD  CONSTRAINT [PK_SimpleTable_Ident]
PRIMARY KEY CLUSTERED (Ident);
CREATE NONCLUSTERED INDEX ix_SimpleTable_ID
    ON Examples.SimpleTable(ID);
```

```
GO
CREATE PROCEDURE Examples.GetValues
    @PARAMETER1 INT
AS
    SELECT
        ID,
        Value
    FROM Examples.SimpleTable
    WHERE
        ID = @PARAMETER1;
GO

ALTER DATABASE ExamBook762Ch4_QueryStore
SET QUERY_STORE = ON (
    INTERVAL_LENGTH_MINUTES = 1
);

EXEC Examples.GetValues 1;
GO 20
```

```
SELECT
    p.plan_id,
    p.query_id,
    q.object_id,
    force_failure_count,
    last_force_failure_reason_desc
FROM sys.query_store_plan AS p
INNER JOIN sys.query_store_query AS q
    ON p.query_id = q.query_id
WHERE is_forced_plan = 1;
```

```
CREATE DATABASE ExamBook762Ch4_QueryPlans;
GO
USE ExamBook762Ch4_QueryPlans;
GO
CREATE SCHEMA Examples;
GO
CREATE TABLE Examples.OrderLines (
    OrderLineID int NOT NULL,
    OrderID int NOT NULL,
    StockItemID int NOT NULL,
    Description nvarchar(100) NOT NULL,
    PackageTypeID int NOT NULL,
    Quantity int NOT NULL,
    UnitPrice decimal(18, 2) NULL,
    TaxRate decimal(18, 3) NOT NULL,
    PickedQuantity int NOT NULL,
    PickingCompletedWhen datetime2(7) NULL,
    LastEditedBy int NOT NULL,
    LastEditedWhen datetime2(7) NOT NULL);
GO
INSERT INTO Examples.OrderLines
SELECT *
FROM WideWorldImporters.Sales.OrderLines;
GO
CREATE INDEX ix_OrderLines_StockItemID
ON Examples.OrderLines (StockItemID);
GO
```

```
SET SHOWPLAN_XML ON;
GO
BEGIN TRANSACTION;
    UPDATE Examples.OrderLines
        SET StockItemID = 300
        WHERE StockItemID < 100;
    SELECT
        OrderID,
        Description,
        UnitPrice
    FROM Examples.OrderLines
    WHERE StockItemID = 300;
ROLLBACK TRANSACTION;
GO
SET SHOWPLAN_XML OFF;
GO
```

```
SET STATISTICS XML ON;
GO
BEGIN TRANSACTION;
    UPDATE Examples.OrderLines
        SET StockItemID = 300
        WHERE StockItemID < 100;
    SELECT
        OrderID,
        Description,
        UnitPrice
    FROM Examples.OrderLines
    WHERE StockItemID = 300;
ROLLBACK TRANSACTION;
GO
SET STATISTICS XML OFF;
GO
```

```
SELECT
    c.LastName,
    c.FirstName,
    c.CompanyName,
    year(OrderDate) AS OrderYear,
    sum(OrderQty) AS OrderQty,
    p.Name AS ProductName,
    sum(LineTotal) AS SalesTotal
FROM SalesLT.SalesOrderHeader soh
JOIN SalesLT.SalesOrderDetail sod ON
    soh.SalesOrderID = sod.SalesOrderID
JOIN SalesLT.Customer c ON
    soh.CustomerID = c.CustomerID
JOIN SalesLT.Product p ON
    sod.ProductID = p.ProductID
GROUP BY
    c.LastName,
    c.FirstName,
    c.CompanyName,
    year(OrderDate),
    p.Name
ORDER BY
    c.CompanyName,
    c.LastName,
    c.FirstName,
    p.Name;
GO 20
```

```
ALTER RESOURCE GOVERNOR RECONFIGURE;
GO
```

```
CREATE RESOURCE POOL poolExamBookDaytime
WITH (
    MIN_CPU_PERCENT = 50,
    MAX_CPU_PERCENT = 80,
    CAP_CPU_PERCENT = 90,
    AFFINITY SCHEDULER = (0 TO 3),
    MIN_MEMORY_PERCENT = 50,
    MAX_MEMORY_PERCENT = 100,
    MIN_IOPS_PER_VOLUME = 20,
    MAX_IOPS_PER_VOLUME = 100
);
GO
CREATE RESOURCE POOL poolExamBookNighttime
WITH (
    MIN_CPU_PERCENT =     0,
    MAX_CPU_PERCENT = 50,
    CAP_CPU_PERCENT = 50,
    AFFINITY SCHEDULER = (0 TO 3),
    MIN_MEMORY_PERCENT = 5,
    MAX_MEMORY_PERCENT = 15,
    MIN_IOPS_PER_VOLUME = 45,
    MAX_IOPS_PER_VOLUME = 100
);
GO
ALTER RESOURCE GOVERNOR RECONFIGURE;
GO
```

```
CREATE WORKLOAD GROUP apps
WITH (
    IMPORTANCE = HIGH,
    REQUEST_MAX_MEMORY_GRANT_PERCENT = 35,
    REQUEST_MAX_CPU_TIME_SEC = 0, --0 = unlimited
    REQUEST_MEMORY_GRANT_TIMEOUT_SEC = 60, --seconds
    MAX_DOP = 0, -- uses global setting
    GROUP_MAX_REQUESTS = 1000 --0 = unlimited
)
USING "poolExamBookNighttime";
GO
CREATE WORKLOAD GROUP reports
WITH (
    IMPORTANCE = LOW,
    REQUEST_MAX_MEMORY_GRANT_PERCENT = 25,
    REQUEST_MAX_CPU_TIME_SEC = 0, --0 = unlimited
    REQUEST_MEMORY_GRANT_TIMEOUT_SEC = 60, --seconds
    MAX_DOP = 0, -- uses global setting
    GROUP_MAX_REQUESTS = 100 --0 = unlimited
)
USING "poolExamBookNighttime";
GO
ALTER RESOURCE GOVERNOR RECONFIGURE;
GO
```

```
USE master
GO
CREATE TABLE tblClassificationTime  (
    TimeOfDay SYSNAME NOT NULL,
    TimeStart TIME NOT NULL,
    TimeEnd   TIME NOT NULL
) ;
GO
INSERT INTO tblClassificationTime
VALUES('apps', '8:00 AM', '6:00 PM');
GO
INSERT INTO tblClassificationTime
VALUES('reports', '6:00 PM', '8:00 AM');
GO
```

```
USE master;
GO
CREATE FUNCTION fnTimeOfDayClassifier()
RETURNS sysname
WITH SCHEMABINDING  AS
BEGIN
    DECLARE @TimeOfDay sysname
    DECLARE @loginTime time
    SET @loginTime = CONVERT(time,GETDATE())
    SELECT
        TOP 1 @TimeOfDay = TimeOfDay
    FROM dbo.tblClassificationTime
    WHERE TimeStart <= @loginTime and TimeEnd >= @loginTime
    IF(@TimeOfDay IS NOT NULL)
        BEGIN
            RETURN @TimeOfDay
        END
    RETURN N'default'
END;
GO
ALTER RESOURCE GOVERNOR with (CLASSIFIER_FUNCTION = dbo.fnTimeOfDayClassifier);
ALTER RESOURCE GOVERNOR RECONFIGURE;
GO
```

```
--Current runtime data
SELECT * FROM sys.dm_resource_governor_resource_pools;
GO

SELECT * FROM sys.dm_resource_governor_workload_groups;
GO

--Determine the workload group for each session
SELECT
    s.group_id,
    CAST(g.name as nvarchar(20)) AS WkGrp,
    s.session_id,
    s.login_time,
    CAST(s.host_name as nvarchar(20)) AS Host,
    CAST(s.program_name AS nvarchar(20))  AS Program
FROM sys.dm_exec_sessions s
INNER JOIN sys.dm_resource_governor_workload_groups g
    ON g.group_id = s.group_id
ORDER BY g.name ;
GO

SELECT
    r.group_id,
    g.name,
    r.status,
    r.session_id,
    r.request_id,
    r.start_time,
    r.command,
    r.sql_handle,
    t.text
```

```
FROM sys.dm_exec_requests r
INNER JOIN sys.dm_resource_governor_workload_groups g
    ON g.group_id = r.group_id
CROSS APPLY sys.dm_exec_sql_text(r.sql_handle) AS t
    ORDER BY g.name
GO

-- Determine the classifier running the request
SELECT
    s.group_id,
    g.name,
    s.session_id,
    s.login_time,
    s.host_name,
    s.program_name
FROM sys.dm_exec_sessions s
INNER JOIN sys.dm_resource_governor_workload_groups g
    ON g.group_id = s.group_id  AND
        s.status = 'preconnect'
ORDER BY g.name;
GO

SELECT
    r.group_id,
    g.name,
    r.status,
    r.session_id,
    r.request_id,
    r.start_time,
    r.command,
    r.sql_handle,
    t.text
```

```
FROM sys.dm_exec_requests r
INNER JOIN sys.dm_resource_governor_workload_groups g
    ON g.group_id = r.group_id
        AND r.status = 'preconnect'
CROSS APPLY sys.dm_exec_sql_text(r.sql_handle) AS t
ORDER BY g.name;
GO
```

```
ALTER DATABASE <databasename>
SET OFFLINE;
GO
ALTER DATABASE <databasename>
MODIFY FILE (NAME = <databasename>_Data, FILENAME = "<drive:\filepath>\
Data\<databasename>_Data.mdf");
GO
ALTER DATABASE <databasename>
MODIFY FILE (NAME = <databasename>_Log, FILENAME = "drive:filepath>\
Log\<databasename>_Log.mdf");
ALTER DATABASE <databasename>
SET ONLINE;
```

```
--Create a database on 4 drives
CREATE DATABASE DB1 ON
PRIMARY
    (Name = <databasename>, FILENAME = '<drive1:filepath>\<databasename>.mdf'),
    FILEGROUP FGHeavyAccess1
    (Name = <databasename>_1, FILENAME = '<drive3:filepath>\<databasename>_1.ndf')
LOG ON
    (Name = <databasename>_1_Log, FILENAME = '<drive3:filepath>\<databasename>_1_
log.ldf'),
    (Name = <databasename>_1, FILENAME = '<drive4:filepath>\<databasename>_1_
log_2.ldf');
-- Add filegroup for index
ALTER DATABASE <databasename>
    ADD FILEGROUP FGIndex;
--  Add data file to the new filegroup
ALTER DATABASE <databasename>
ADD FILE (
    NAME = <databasename>,
    FILENAME = '<drive1:filepath>\<databasename>.ndf',
    SIZE=1024MB,
    MAXSIZE=10GB,
    FILEGROWTH=10%)
TO FILEGROUP FGIndex;
-- Add index to filegroup
CREATE NONCLUSTERED INDEX ix_Example
    ON Examples.BusyTable(TableColumn)
   ON FGIndex;
```

```
CREATE PARTITION SCHEME PSYear
    AS PARTITION PFYearRange
    TO (FGYear1, FGYear2, FGYear3, FGYear4);
```

```
EXEC sp_configure 'show advanced options', 1;
GO
RECONFIGURE;
GO
EXEC sp_configure 'min memory per query', 512 ;
GO
RECONFIGURE;
GO
```

```
SELECT
    object_name,
    counter_name,
    instance_name,
    cntr_value,
    cntr_type
FROM sys.dm_os_performance_counters
WHERE object_name = 'SQLServer:Buffer Manager' AND
    counter_name IN
        ('Page lookups/sec', 'Page reads/sec', 'Page writes/sec')
```

```
SELECT
      Event_Category,
      Event_Type,
      Event_Subtype_Desc,
      Event_Count,
      Description,
      Start_Time
FROM sys.event_log
WHERE Event_Type = 'deadlock' OR
      Event_Type like 'throttling%'
```

```
ADD TARGET package0.event_file(SET filename=N'C:\ExamBook762Ch4\query.xel',
    max_file_size=(5),max_rollover_files=(4)),
ADD TARGET package0.ring_buffer
```

```
ADD EVENT sqlserver.sql_statement_completed(
    ACTION (sqlserver.session_id,
        sqlserver.sql_text))
```

```
CREATE EVENT SESSION [stored_proc]
ON SERVER
ADD EVENT sqlserver.sp_statement_completed(
    ACTION (sqlserver.session_id,
        sqlserver.sql_text))
ADD TARGET package0.event_file(SET filename=N'C:\ExamBook762Ch4\query.xel',
    max_file_size=(5),max_rollover_files=(4)),
ADD TARGET package0.ring_buffer;
```

```
idx_id ObjectName                                index_type_desc             pg_ct
AvgPageSpacePct     frag_ct  AvgFragPct
-------- ------------------------------------ ------------------------------
------- ------------------------- ---------- --------------------------
1          PK_SalesOrders_OrderD        CLUSTERED INDEX           2037
54.9851742031134 237        34.02189781021898
2          IX_SalesOrders_CustomerID NONCLUSTERED INDEX   685    98.4313442055844
2             0
```

Table of Contents

Title Page 2
Copyright Page 3
Contents at a glance 5
Contents 6
Introduction 10
Organization of this book 10
Microsoft certifications 11
Acknowledgments 11
Free ebooks from Microsoft Press 12
Microsoft Virtual Academy 12
Quick access to online references 12
Errata, updates, & book support 12
We want to hear from you 13
Stay in touch 13
Important: How to use this book to study for the exam 13
Chapter 1. Design and implement database objects 14
Skill 1.1: Design and implement a relational database schema 14
Designing tables and schemas based on business requirements 15
Improving the design of tables by using normalization 17
Writing table create statements 25
Determining the most efficient data types to use 30
Skill 1.2: Design and implement indexes 40
Design new indexes based on provided tables, queries, or plans 42
Distinguish between indexed columns and included columns 61
Implement clustered index columns by using best practices 67
Recommend new indexes based on query plans 69
Skill 1.3: Design and implement views 74
Design a view structure to select data based on user or business requirements 75
Identify the steps necessary to design an updateable view 81
Implement partitioned views 89
Implement indexed views 93
Skill 1.4: Implement columnstore indexes 96

Determine use cases that support the use of columnstore indexes 96
Identify proper usage of clustered and non-clustered columnstore indexes 99
Design standard non-clustered indexes in conjunction with clustered columnstore indexes 114
Implement columnstore index maintenance 119
Chapter summary 126
Thought experiment 128
Thought experiment answer 130
Chapter 2. Implement programmability objects 133
Skill 2.1 Ensure data integrity with constraints 134
Define table and foreign-key constraints to enforce business rules 134
Write Transact-SQL statements to add constraints to tables 155
Identify results of Data Manipulation Language (DML) statements given existing tables and constraints 160
Identify proper usage of PRIMARY KEY constraints 162
Skill 2.2 Create stored procedures 168
Design stored procedure components and structure based on business requirements 169
Implement input and output parameters 174
Implement table-valued parameters 177
Implement return codes 180
Streamline existing stored procedure logic 181
Implement error handling and transaction control logic within stored procedures 186
Skill 2.3 Create triggers and user-defined functions 206
Design trigger logic based on business requirements 206
Determine when to use Data Manipulation Language (DML) triggers, Data Definition Language (DDL) triggers, or logon triggers 219
Recognize results based on execution of AFTER or INSTEAD OF triggers 229
Design scalar-valued and table-valued user-defined functions based on business requirements 233
Identify differences between deterministic and non-deterministic functions 241
Chapter summary 244
Thought Experiment 246
Though Experiment Answer 246
Chapter 3. Manage database concurrency 252

Skill 3.1: Implement transactions 252
Identify DML statement results based on transaction behavior 253
Recognize differences between and identify usage of explicit and implicit transactions 261
Implement savepoints within transactions 268
Determine the role of transactions in high-concurrency databases 271
Skill 3.2: Manage isolation levels 276
Identify differences between isolation levels 277
Define results of concurrent queries based on isolation level 279
Identify the resource and performance impact of given isolation levels 290
Skill 3.3: Optimize concurrency and locking behavior 292
Troubleshoot locking issues 292
Identify lock escalation behaviors 300
Capture and analyze deadlock graphs 300
Identify ways to remediate deadlocks 304
Skill 3.4: Implement memory-optimized tables and native stored procedures 306
Define use cases for memory-optimized tables 306
Optimize performance of in-memory tables 309
Determine best case usage scenarios for natively compiled stored procedures 321
Enable collection of execution statistics for natively compiled stored procedures 322
Chapter summary 326
Thought experiment 329
Thought experiment answers 330
Chapter 4. Optimize database objects and SQL infrastructure 332
Skill 4.1: Optimize statistics and indexes 332
Determine the accuracy of statistics and the associated impact to query plans and performance 333
Design statistics maintenance tasks 341
Use dynamic management objects to review current index usage and identify missing indexes 345
Consolidate overlapping indexes 351
Skill 4.2: Analyze and troubleshoot query plans 353
Capture query plans using extended events and traces 354
Identify poorly performing query plan operators 364
Compare estimated and actual query plans and related metadata 390

Configure Azure SQL Database Performance Insight 396
Skill 4.3: Manage performance for database instances 404
Manage database workload in SQL Server 404
Design and implement Elastic Scale for Azure SQL Database 412
Select an appropriate service tier or edition 415
Optimize database file and tempdb configuration 417
Optimize memory configuration 421
Monitor and diagnose schedule and wait statistics using dynamic management objects 422
Troubleshoot and analyze storage, IO, and cache issues 425
Monitor Azure SQL Database query plans 428
Skill 4.4: Monitor and trace SQL Server baseline performance metrics 429
Monitor operating system and SQL Server performance metrics 430
Compare baseline metrics to observed metrics while troubleshooting performance issues 435
Identify differences between performance monitoring and logging tools 438
Monitor Azure SQL Database performance 439
Determine best practice use cases for extended events 444
Distinguish between Extended Events targets 444
Compare the impact of Extended Events and SQL Trace 445
Define differences between Extended Events Packages, Targets, Actions, and Sessions 445
Chapter summary 447
Thought experiment 450
Thought experiment answer 452
Index 454
About the authors 483
Free ebooks 484
Survey 486
Code Snippets 488