Microsoft

Developing SQL Databases

Exam Ref 70-762

Louis Davidson
Stacia Varga

FREE SAMPLE CHAPTER

SHARE WITH OTHERS

Microsoft

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.

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

Library of Congress Control Number: 2016962647

First Printing January 2017

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

This page intentionally left blank

Contents

This page intentionally left blank

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.

This page intentionally left blank

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 success-

fully 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

```
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.

```
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:

```
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:

```
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:

```
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:

```
      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.

```
(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:

```
      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:

```
      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:

```
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:

```
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:

```
SET IMPLICIT_TRANSACTIONS ON;
```

Next, execute an INSERT statement and then check the status of open transactions:

```
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:

```
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:

```
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

```
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:

```
(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

```
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

```
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:

```
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:

  ```
  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:

```
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.

TABLE 3-1 Lock compatibility for commonly encountered lock modes

	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

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

```
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:

```
SET TRANSACTION ISOLATION LEVEL READ COMMITTED;
```

To test the behavior of the READ COMMITTED isolation level, execute the following statements:

```
BEGIN TRANSACTION;
    UPDATE Examples.IsolationLevels
        SET ColumnText = 'Row 1 Updated'
        WHERE RowId = 1;
```

In a new session, read the table that you just updated:

```
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:

```
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:

```
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:

```
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:

```
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:

```
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:

```
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:

```
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:

```
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:

```
SET TRANSACTION ISOLATION LEVEL SERIALIZABLE;
```

Then create a new session to insert another row:

```
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:

```
ALTER DATABASE ExamBook762Ch3
SET ALLOW_SNAPSHOT_ISOLATION ON;
```

Now set the isolation level for the session and start a transaction:

```
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:

```
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

```
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:

```
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:

```
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:

```
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:

```
CREATE INDEX Ix_RowId ON Examples.IsolationLevels (RowId);
```

Next set up a new transaction:

```
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:

```
ALTER INDEX Ix_RowId
    ON Examples.IsolationLevels REBUILD;
```

SQL Server returns the following error due to the metadata change:

```
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:

```
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:

```
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:

```
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:

```
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:

```
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

```
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.

TABLE 3-2 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).

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:

```
BEGIN TRANSACTION;
    SELECT RowId, ColumnText
    FROM Examples.LockingA
    WITH (HOLDLOCK, ROWLOCK);
```

In a separate session, start another transaction:

```
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:

```
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.

```
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
```

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:

```
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:

```
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.

TABLE 3-3 sys.dm_os_waiting_tasks

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).

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

```
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.

```
res_typ  res_dbid   res_entid            mode  s_id   blocking_s_id
-------  --------   ------------------   ----  ------  -----------------
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.

TABLE 3-4 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).

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:

```
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.

```
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:

```
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:

```
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:

```
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:

```
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:

```
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:

```
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.

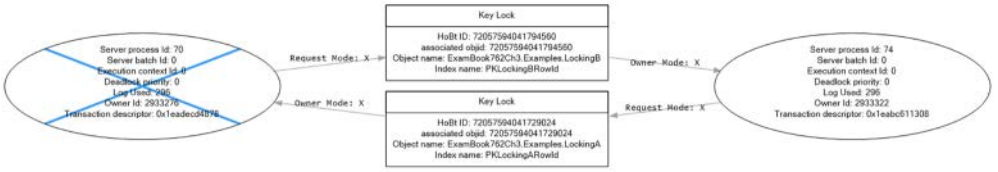

FIGURE 3-1 A deadlock graph

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.

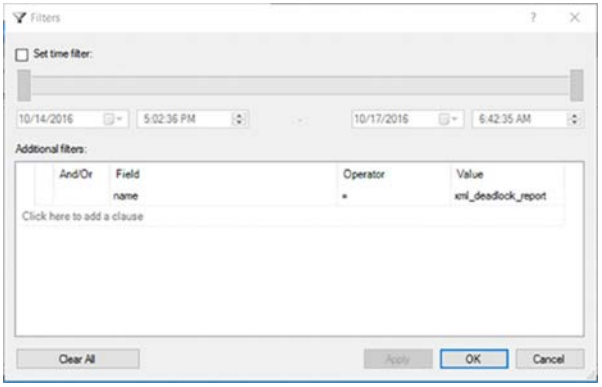

FIGURE 3-2 An Extended Events filter for xml_deadlock_report

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

```
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-opti-

mized 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

```
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

```
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

```
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

```
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:

```
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
```

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.

```
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.

```
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.

```
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

```
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:

```
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:

```
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

```
--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

```
--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

```
--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

```
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:

```
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
```

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

```
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.

```
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)
```

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.

```
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:

```
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?

```
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.

This page intentionally left blank

Index

A

B

C

D

E

F

G

H

I

J

K

L

M

N

O

P

Q

R

S

T

U

V

W

X

Y