Microsoft

Querying Data with Transact-SQL

Exam Ref 70-761

Itzik Ben-Gan

Microsoft

Exam Ref 70-761 Querying Data with Transact-SQL

Itzik Ben-Gan

Exam Ref 70-761 Querying Data with Transact-SQL

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

ISBN-13: 978-1-5093-0433-2
ISBN-10: 1-5093-0433-9

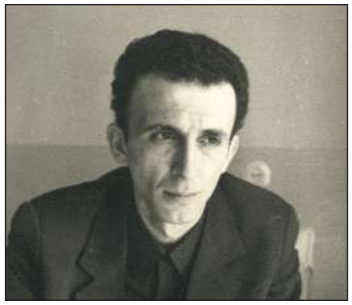
Library of Congress Control Number: 2017935711

First Printing April 2017

Editor-in-Chief	Greg Wiegand
Acquisitions Editor	Trina MacDonald
Development Editor	Troy Mott
Managing Editor	Sandra Schroeder
Senior Project Editor	Tracey Croom
Editorial Production	Backstop Media
Copy Editor	Christina Rudloff
Indexer	Julie Grady
Proofreader	Christina Rudloff
Technical Editor	Dejan Sarka
Cover Designer	Twist Creative, Seattle

In memory of my dad, Gabriel Ben-Gan, who appreciated the beauty of numbers, logic and puzzles.

—Itzik



This page intentionally left blank

Contents at a glance

This page intentionally left blank

Contents

This page intentionally left blank

Introduction

The 70-761 exam focuses on T-SQL querying and programming constructs. Whether you're taking it as part of a Microsoft data platform related certification path, or to assess your T-SQL skills for any other reason, this is a crucial exam, since it covers the essential language constructs. If you need to do work with any of the Microsoft data platform technologies, a good grasp of T-SQL is vital.

The exam covers query constructs like filtering, grouping and sorting, combining data from multiple tables using joins, subqueries, set operators, modifying data, as well as some aspects of data definition like choosing data types and enforcing data integrity. The exam also covers more advanced query constructs like pivoting and unpivoting data, using window functions, grouping sets, using the APPLY operator, the complexities of NULLs, as well as implicit conversions. The exam covers querying system-versioned temporal tables, as well as XML and JSON data. The exam also covers T-SQL modules and programmatic constructs like views, user-defined functions, and stored procedures, as well as working with transactions and error handling.

This exam is intended for SQL Server database administrators, system engineers, and developers with two or more years of experience who are seeking to validate their skills and knowledge in writing queries.

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

Itzik Ben-Gan Writing a book is a demanding yet rewarding project. A big part of what makes it rewarding is that you get to work with other people, and together create something that will hopefully contribute to increase technological and scientific knowledge out there. I'd like to recognize those who were involved in this book for their contributions. Special thanks to Trina MacDonald, my editor, for your outstanding handling of the project. Many thanks to Dejan Sarak who tech edited the book, as well as wrote the section about querying XML and JSON data. I know I can always count on you both in terms of your depth of knowledge and ethics. Thanks, are also due to Troy Mott, the book's Development Editor, Christina Rudloff, the Copy Editor, Ellie Volckhausen who handled the book's layout, and Julie Grady, the book's Indexer. Lastly, to Lilach, my wife, for helping with the first reviews, and for giving reason to what I do.

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/exam761transactsql/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/exam761transactsql/errata

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*.

Download the source code and sample database from the book's website

https://aka.ms/exam761transactsql/detail

The author also created a personal companion website for the book, which you can find at *http://tsql.solidq.com/books/er70761*.

In order to run the code samples from the book, you will need access to SQL Server 2016 with Service Pack1 or later, or Azure SQL Database.

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 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 1

Manage data with Transact-SQL

Transact-SQL (T-SQL) is the main language used to manage and manipulate data in Microsoft SQL Server and Azure SQL Database. If you work with any of the Microsoft SQL products—as a developer, DBA, BI professional, data analyst, data scientist, or in any other capacity—you need to know your T-SQL. Exam 70-761 is a foundational exam that tests your T-SQL querying knowledge, and is a required part of several of the Microsoft SQL certification paths.

This chapter focuses on managing data with T-SQL. It covers the elements of the SELECT statement, how to combine data from multiple tables with set operators and joins, use of built-in functions, and how to modify data.

IMPORTANT

Have you read page xv?

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

Skills in this chapter:

- Create Transact-SQL SELECT queries
- Query multiple tables by using joins
- Implement functions and aggregate data
- Modify data

Skill 1.1: Create Transact-SQL SELECT queries

To write correct and robust T-SQL code, it's important to first understand the roots of the language, as well as a concept called logical query processing. You also need to understand the structure of the SELECT statement and how to use its clauses to perform data manipulation tasks like filtering and sorting. You often need to combine data from different sources, and one of the ways to achieve this in T-SQL is using set operators.

This section covers how to:

- Identify proper SELECT query structure
- Write specific queries to satisfy business requirements
- Construct results from multiple queries using set operators
- Distinguish between UNION and UNION ALL behavior
- Identify the query that would return expected results based on provided table structure and/or data

Understanding the foundations of T-SQL

Many aspects of computing, like programming languages, evolve based on intuition and the current trend. Without strong foundations, their lifespan can be very short, and if they do survive, often the changes are very rapid due to changes in trends. T-SQL is different, mainly because it has strong foundations—mathematics. You don't need to be a mathematician to write SQL well (though it certainly doesn't hurt), but as long as you understand what those foundations are, and some of their key principles, you will better understand the language you are dealing with. Without those foundations, you can still write T-SQL code—even code that runs successfully—but it will be like eating soup with a fork!

Evolution of T-SQL

As mentioned, unlike many other aspects of computing, T-SQL is based on strong mathematical foundations. Understanding some of the key principles from those foundations can help you better understand the language you are dealing with. Then you will think in T-SQL terms when coding in T-SQL, as opposed to coding with T-SQL while thinking in procedural terms.

Figure 1-1 illustrates the evolution of T-SQL from its core mathematical foundations.

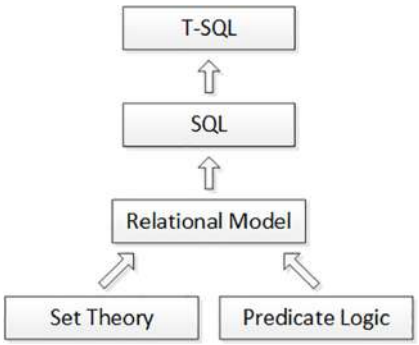


FIGURE 1-1 Evolution of T-SQL

T-SQL is the main language used to manage and manipulate data in the Microsoft relational database management systems (RDBMSs) SQL Server (the box product), and Azure SQL Database (the cloud platform). The code base for both the cloud platform and the box prod-

uct is one unified code base. For simplicity I will use the term *SQL Server* in reference to both, as far as T-SQL is concerned. SQL Server also supports other languages, like Microsoft Visual C# and Microsoft Visual Basic, but T-SQL is usually the preferred language for data management and manipulation.

T-SQL is a dialect of standard SQL. SQL is a standard of both the International Organization for Standards (ISO) and the American National Standards Institute (ANSI). The two standards for SQL are basically the same. The SQL standard keeps evolving with time. Following is a list of the major revisions of the standard so far:

- SQL-86
- SQL-89
- SQL-92
- SQL:1999
- SQL:2003
- SQL:2006
- SQL:2008
- SQL:2011

All leading database vendors, including Microsoft, implement a dialect of SQL as the main language to manage and manipulate data in their database platforms. Therefore, the core language elements look the same. However, each vendor decides which features to implement, and which features to not implement. Also, the standard sometimes leaves some aspects as an implementation choice. Each vendor also usually implements extensions to the standard in cases where the vendor feels that an important feature isn't covered by the standard.

Writing in a standard way is considered a best practice. When you do so, your code is more portable. Your knowledge is more portable, too, because it is easy for you to start working with new platforms. When the dialect you're working with supports both a standard and a nonstandard way to do something, you should always prefer the standard form as your default choice. You should consider a nonstandard option only when it has some important benefit to you that is not covered by the standard alternative.

As an example of when to choose the standard form, T-SQL supports two "not equal to" operators: <> and !=. The former is standard and the latter is not. In this case, the choice should be obvious: go for the standard one!

As an example of when the choice of standard or nonstandard depends on the circumstances, consider the following: T-SQL supports multiple functions that convert a source expression to a target type. Among them are the CAST and CONVERT functions. The former is standard and the latter isn't. The nonstandard CONVERT function has a style argument that CAST doesn't support. Because CAST is standard, you should consider it your default choice for conversions. You should consider using CONVERT only when you need to rely on the style argument.

Yet another example of choosing the standard form is in the termination of T-SQL statements. According to standard SQL, you should terminate your statements with a semicolon.

The T-SQL documentation specifies that not terminating all statements with a semicolon is a deprecated feature, but T-SQL currently doesn't enforce this for all statements, rather only in cases where there would otherwise be ambiguity of code elements. For example, a statement preceding the WITH clause of a common table expression (CTE) has to be terminated because this clause can also be used to define a table hint in the preceding statement. As another example, the MERGE statement has to be terminated due to possible ambiguity of the MERGE keyword. You should still follow the standard and terminate all of your statements even where it is currently not required.

Standard SQL is based on the relational model, which is a mathematical model for data management and manipulation. The relational model was initially created and proposed by Edgar F. Codd in 1969. After its creation, it has been explained and further developed by Codd, Chris Date, Hugh Darwen, and others.

A common misconception is that the name "relational" has to do with relationships between tables (that is, foreign keys). Actually, the true source for the model's name is the mathematical concept relation.

A relation in the relational model is what SQL represents with a table. The two are not synonymous. You could say that a table is an attempt by SQL to represent a relation (in addition to a relation variable, but that's not necessary to get into here). Some say that it is not a very successful attempt. Even though SQL is based on the relational model, it deviates from it in a number of ways. But it's important to note that as you understand the model's principles, you can use SQL—or more precisely, the dialect you are using—in a relational way. More on this, including a further reading recommendation, is in the next section, "Using T-SQL in a relational way."

Getting back to a relation, which is what SQL attempts to represent with a table: a relation has a heading and a body. The heading is a set of attributes (what SQL attempts to represent with columns), each of a given type. An attribute is identified by name and type name. The body is a set of tuples (what SQL attempts to represent with rows). Each tuple's heading is the heading of the relation. Each value of each tuple's attribute is of its respective type.

Some of the most important aspects to understand about T-SQL stem from the relational model's core foundations: set theory and predicate logic.

Remember that the heading of a relation is a set of attributes, and the body is a set of tuples. So what is a set? According to the creator of mathematical set theory, Georg Cantor, a set is described as follows:

> *By a "set" we mean any collection M into a whole of definite, distinct objects m (which are called the "elements" of M) of our perception or of our thought.*
>
> *—Georg Cantor, in "Georg Cantor" by Joseph W. Dauben (Princeton University Press, 1990)*

There are a number of very important elements in this definition that, if understood, should have direct implications on your T-SQL coding practices. One element that requires notice is the term *whole*. A set should be considered as a whole. This means that you do not interact with the individual elements of the set, rather with the set as a whole.

Notice the term *distinct*; a set has no duplicates. Codd once remarked on the no duplicates aspect: "If something is true, then saying it twice won't make it any truer." For example, the set `{a, b, c}` is considered equal to the set `{a, a, b, c, c, c}`.

Another critical aspect of a set doesn't explicitly appear in the aforementioned definition by Cantor, but rather is implied; there's no relevance to the order of elements in a set. In contrast, a sequence (which is an ordered set), does have an order to its elements. Combining the no duplicates and no relevance to order aspects means that the collection `{a, b, c}` is a set, but the collection `{b, a, c, c, a, c}` isn't.

The other branch of mathematics that the relational model is based on is called *predicate logic*. A predicate is an expression that when attributed to some object, makes a proposition either true or false. For example, "salary greater than $50,000" is a predicate. You can evaluate this predicate for a specific employee, in which case you have a proposition. For example, suppose that for a particular employee, the salary is $60,000. When you evaluate the proposition for that employee, you get a true proposition. In other words, a predicate is a parameterized proposition.

The relational model uses predicates as one of its core elements. You can enforce data integrity by using predicates. You can filter data by using predicates. You can even use predicates to define the data model itself. You first identify propositions that need to be stored in the database. Here's an example proposition: an order with order ID 10248 was placed on February 12, 2017 by the customer with ID 7, and handled by the employee with ID 3. You then create predicates from the propositions by removing the data and keeping the heading. Remember, the heading is a set of attributes, each identified by name and type name. In this example, you have orderid INT, orderdate DATE, custid INT, and empid INT.

Using T-SQL in a relational way

As mentioned, T-SQL is based on SQL, which in turn is based on the relational model. However, there are a number of ways in which SQL, and therefore T-SQL, deviates from the relational model. But T-SQL gives you enough tools so that if you understand the relational model, you can use the language in a relational manner, and thus write more correct code.

> ***MORE INFO*** **SQL AND RELATIONAL THEORY**
>
> **For detailed information about the differences between SQL and the relational model, and how to use SQL in a relational way, see *SQL and Relational Theory*, 3rd Edition by C. J. Date (O'Reilly Media, 2015). It's an excellent book that all database practitioners should read.**

Remember that a relation has a heading and a body. The heading is a set of attributes and the body is a set of tuples. Remember that a set is supposed to be considered as a whole.

What this translates to in T-SQL is that you're supposed to write queries that interact with the tables as a whole. You should try to avoid using iterative constructs like cursors and loops that iterate through the rows one at a time. You should also try to avoid thinking in iterative terms because this kind of thinking is what leads to iterative solutions.

For people with a procedural programming background, the natural way to interact with data (in a file, record set, or data reader) is with iterations. So using cursors and other iterative constructs in T-SQL is, in a way, an extension to what they already know. However, the correct way from the relational model's perspective is not to interact with the rows one at a time, rather, use relational operations and return a relational result. This, in T-SQL, translates to writing queries.

Remember also that a set has no duplicates. In other words, it has unique members. T-SQL doesn't always enforce this rule. For example, you can create a table without a key. In such a case, you are allowed to have duplicate rows in the table. To follow relational theory, you need to enforce uniqueness in your tables. For example, you can enforce uniqueness in your tables by using a primary key, or a unique constraint.

Even when the table doesn't allow duplicate rows, a query against the table can still return duplicate rows in its result. Consider the following query:

```
USE TSQLV4;

SELECT country
FROM HR.Employees;
```

The query is issued against the TSQLV4 sample database. It returns the country attribute of the employees stored in the HR.Employees table. According to the relational model, a relational operation against a relation is supposed to return a relation. In this case, this should translate to returning the *set* of countries where there are employees, with an emphasis on set, as in no duplicates. However, T-SQL doesn't attempt to remove duplicates by default.

Here's the output of this query:

```
country
---------------
USA
USA
USA
USA
UK
UK
UK
USA
UK
```

In fact, T-SQL is based more on multiset theory than on set theory. A *multiset* (also known as a bag or a superset) in many respects is similar to a set, but can have duplicates. As mentioned, T-SQL does give you enough tools so that if you want to follow relational theory, you can do so. For example, the language provides you with a DISTINCT clause to remove duplicates. Here's the revised query:

```
SELECT DISTINCT country
FROM HR.Employees;
```

Here's the revised query's output:

```
Country
---------------
UK
USA
```

Another fundamental aspect of a set is that there's no relevance to the order of the elements. For this reason, rows in a table have no particular order, conceptually. So when you issue a query against a table and don't indicate explicitly that you want to return the rows in particular presentation order, the result is supposed to be relational. Therefore, you shouldn't assume any specific order to the rows in the result; never mind what you know about the physical representation of the data, for example, when the data is indexed.

As an example, consider the following query:

```
SELECT empid, lastname
FROM HR.Employees;
```

When this query was run on one system, it returned the following output, which looks like it is sorted by the column lastname:

```
empid       lastname
----------- --------------------
8           Cameron
1           Davis
9           Doyle
2           Funk
7           King
3           Lew
5           Mortensen
4           Peled
6           Suurs
```

Even if the rows were returned in a different order, the result would have still been considered correct. SQL Server can choose between different physical access methods to process the query, knowing that it doesn't need to guarantee the order in the result. For example, SQL Server could decide to parallelize the query or scan the data in file order (as opposed to index order).

If you do need to guarantee a specific presentation order to the rows in the result, you need to add an ORDER BY clause to the query, as follows:

```
SELECT empid, lastname
FROM HR.Employees
ORDER BY empid;
```

This time, the result isn't relational, it's what standard SQL calls a *cursor*. The order of the rows in the output is guaranteed based on the empid attribute. Here's the output of this query:

```
empid       lastname
----------- --------------------
1           Davis
2           Funk
3           Lew
4           Peled
5           Mortensen
6           Suurs
7           King
8           Cameron
9           Doyle
```

The heading of a relation is a set of attributes; as such, the attributes are unordered and unique. This means that you are supposed to identify an attribute by name and type name. Conversely, T-SQL does keep track of ordinal positions of columns based on their order of appearance in the table definition. When you issue a query with SELECT *, you are guaranteed to get the columns in the result based on definition order. Also, T-SQL allows referring to ordinal positions of columns from the result in the ORDER BY clause, as follows:

```
SELECT empid, lastname
FROM HR.Employees
ORDER BY 1;
```

Beyond the fact that this practice is not relational, think about the potential for error if at some point you change the SELECT list and forget to change the ORDER BY list accordingly. Therefore, the recommendation is to always indicate the names of the attributes that you need to order by.

T-SQL has another deviation from the relational model in that it allows defining result columns based on an expression without assigning a name to the target column. For example, the following query is valid in T-SQL:

```
SELECT empid, firstname + ' ' + lastname
FROM HR.Employees;
```

This query generates the following output:

```
empid
----------- ------------------------------
1           Sara Davis
2           Don Funk
3           Judy Lew
4           Yael Peled
5           Sven Mortensen
6           Paul Suurs
7           Russell King
8           Maria Cameron
9           Patricia Doyle
```

But according to the relational model, all attributes must have names. In order for the query to be relational, you need to assign an alias to the target attribute. You can do so by using the AS clause, as follows:

```
SELECT empid, firstname + ' ' + lastname AS fullname
FROM HR.Employees;
```

Also, with T-SQL a query can return multiple result columns with the same name. For example, consider a join between two tables, T1 and T2, both with a column called keycol. With T-SQL, a SELECT list can look like the following:

```
SELECT T1.keycol, T2.keycol ...
```

For the result to be relational, all attributes must have unique names, so you would need to use different aliases for the result attributes as follows:

```
SELECT T1.keycol AS key1, T2.keycol AS key2 ...
```

As for predicates, following the law of excluded middle in mathematical logic, a predicate can evaluate to true or false. In other words, predicates are supposed to use two-valued logic. However, Codd wanted to reflect the possibility for values to be missing in his model. He referred to two kinds of missing values: missing but applicable (A-Values marker) and missing but inapplicable (I-Values marker). As an example for a missing but applicable case, consider a mobilephone attribute of an employee. Suppose that an employee has a mobile phone, but did not want to provide this information, for example, for privacy reasons. As an example for a missing but inapplicable case, consider a salescommission attribute of an employee. This attribute is applicable only to sales people, but not to other kinds of employees. According to Codd, a language based on his model should provide two different markers for the two cases. T-SQL—again, based on standard SQL—implements only one general-purpose marker called NULL for any kind of missing value. This leads to three-valued predicate logic. Namely, when a predicate compares two values, for example, mobilephone = '(425) 555-0136', if both are present, the result evaluates to either true or false. But if at least one of them is NULL, the result evaluates to a third logical value—unknown. That's the case both when you use the equality operator = and when you use an inequality operator such as: <>, >, >=, <, <=.

Note that some believe that a valid relational model should follow two-valued logic, and strongly object to the concept of NULLs in SQL. But as mentioned, the creator of the relational model believed in the idea of supporting missing values, and predicates that extend beyond two-valued logic. What's important from a perspective of coding with T-SQL is to realize that if the database you are querying supports NULLs, their treatment is far from being trivial. That is, you need to carefully understand what happens when NULLs are involved in the data you're manipulating with various query constructs, like filtering, sorting, grouping, joining, or intersecting. Hence, with every piece of code you write with T-SQL, you want to ask yourself whether NULLs are possible in the data you're interacting with. If the answer is yes, you want to make sure that you understand the treatment of NULLs in your query, and ensure that your tests address treatment of NULLs specifically.

Using correct terminology

Your use of terminology reflects on your knowledge. Therefore, you should make an effort to understand and use correct terminology. When discussing T-SQL–related topics, people often use incorrect terms. And if that's not enough, even when you do realize what the correct terms are, you also need to understand the differences between the terms in T-SQL and those in the relational model.

As an example of incorrect terms in T-SQL, people often use the terms "field" and "record" to refer to what T-SQL calls "column" and "row," respectively. Fields and records are physical. Fields are what you have in user interfaces in client applications, and records are what you have in files and cursors. Tables are logical, and they have logical rows and columns.

Another example of an incorrect term is referring to "NULL values." A NULL is a marker for a missing value—not a value itself. Hence, the correct usage of the term is either just "NULL" or "NULL marker." Personally, I prefer the former.

Besides using correct T-SQL terminology, it's also important to understand the differences between T-SQL terms and their relational counterparts. Remember from the previous section that T-SQL attempts to represent a relation with a table, a tuple with a row, and an attribute with a column; but the T-SQL concepts and their relational counterparts differ in a number of ways. As long as you are conscious of those differences, you can, and should, strive to use T-SQL in a relational way.

Understanding logical query processing

T-SQL has both logical and physical sides to it. The logical side is the conceptual interpretation of the query that explains what the correct result of the query is. The physical side is the processing of the query by the database engine. Physical processing must produce the result defined by logical query processing. To achieve this goal, the database engine can apply optimization. Optimization can rearrange steps from logical query processing or remove steps altogether, but only as long as the result remains the one defined by logical query processing. The focus of this section is logical query processing—the conceptual interpretation of the query that guarantees returning what I defined as the correct result.

T-SQL as a declarative English-like language

T-SQL, being based on standard SQL, is a declarative English-like language. In this language, declarative means you define what you want, as opposed to imperative languages that define also how to achieve what you want. Standard SQL describes the logical interpretation of the declarative request (the "what" part), but it's the database engine's responsibility to figure out how to physically process the request (the "how" part).

For this reason, it is important not to draw any performance-related conclusions from what you are reviewing about logical query processing. That's because logical query processing only defines the correctness of the query. When addressing performance aspects of the query, you need to understand how optimization works. As mentioned, optimization can be

quite different from logical query processing because it's allowed to change things as long as the result achieved is the one defined by logical query processing.

It's interesting to note that the standard language SQL wasn't originally called so; rather, it was called SEQUEL; an acronym for "structured English query language." But then due to a trademark dispute with an airline company, the language was renamed to SQL, for "structured query language." Still, the point is that you provide your instructions in an English-like manner. For example, consider the instruction, "Bring me a soda from the refrigerator." Observe that in the English instruction, the object comes before the location. Consider the following request in T-SQL:

```
SELECT shipperid, phone, companyname
FROM Sales.Shippers;
```

Observe the similarity of the query's keyed-in order to English. The query first indicates the SELECT list with the attributes you want to return, and then the FROM clause with the table you want to query.

Now try to think of the order in which the request needs to be logically interpreted. For example, how would you define the instructions to a robot instead of a human? The original English instruction to get a soda from the refrigerator would probably need to be revised to something like, "Go to the refrigerator; open the door; get a soda; bring it to me."

Similarly, the logical processing of a query must first know which table is being queried before it can know which attributes can be returned from that table. Therefore, contrary to the keyed-in order of the previous query, the logical query processing has to be as follows:

```
FROM Sales.Shippers
SELECT shipperid, phone, companyname
```

This is a basic example with just two query clauses. Of course, things can get more complex. If you understand the concept of logical query processing well, you will be able to explain many things about the way the language behaves—things that are very hard to explain otherwise.

Logical query processing phases

This section covers logical query processing and the phases involved. The main statement used to retrieve data in T-SQL is the SELECT statement. Following are the main query clauses specified in the order that you are supposed to type them (known as "keyed-in order"):

1. SELECT
2. FROM
3. WHERE
4. GROUP BY
5. HAVING
6. ORDER BY

But as mentioned, the logical query processing order, which is the conceptual interpretation order, is different. It starts with the FROM clause. Here is the logical query processing order of the six main query clauses:

1. FROM
2. WHERE
3. GROUP BY
4. HAVING
5. SELECT
6. ORDER BY

Each phase operates on one or more tables as inputs and returns a virtual table as output. The output table of one phase is considered the input to the next phase. This is in accord with operations on relations that yield a relation. Note that if an ORDER BY is specified, the result isn't relational. This means that you can't operate on such result with an outer query because an outer query expects a relation as input.

Consider the following query as an example:

```
SELECT country, YEAR(hiredate) AS yearhired, COUNT(*) AS numemployees
FROM HR.Employees
WHERE hiredate >= '20140101'
GROUP BY country, YEAR(hiredate)
HAVING COUNT(*) > 1
ORDER BY country, yearhired DESC;
```

This query is issued against the HR.Employees table. It filters only employees that were hired in or after the year 2014. It groups the remaining employees by country and the hire year. It keeps only groups with more than one employee. For each qualifying group, the query returns the hire year and count of employees, sorted by country and hire year, in descending order.

The following sections provide a brief description of what happens in each phase according to logical query processing.

1. EVALUATE THE FROM CLAUSE

In the first phase, the FROM clause is evaluated. That's where you indicate the tables you want to query and table operators like joins if applicable. If you need to query just one table, you indicate the table name as the input table in this clause. Then, the output of this phase is a table result with all rows from the input table. That's the case in the following query: the input is the HR.Employees table (nine rows), and the output is a table result with all nine rows (only a subset of the attributes are shown):

```
empid  country  hiredate
------ -------- ----------
1      USA      2013-05-01
2      USA      2013-08-14
3      USA      2013-04-01
```

```
4       USA       2014-05-03
5       UK        2014-10-17
6       UK        2014-10-17
7       UK        2015-01-02
8       USA       2015-03-05
9       UK        2015-11-15
```

2. FILTER ROWS BASED ON THE WHERE CLAUSE

The second phase filters rows based on the predicate in the WHERE clause. Only rows for which the predicate evaluates to true are returned.

In this query, the WHERE filtering phase filters only rows for employees hired on or after January 1, 2014. Six rows are returned from this phase and are provided as input to the next one. Here's the result of this phase:

```
empid  country  hiredate
------ -------- ----------
4      USA      2014-05-03
5      UK       2014-10-17
6      UK       2014-10-17
7      UK       2015-01-02
8      USA      2015-03-05
9      UK       2015-11-15
```

A typical mistake made by people who don't understand logical query processing is attempting to refer in the WHERE clause to a column alias defined in the SELECT clause. You can't do this because the WHERE clause is evaluated before the SELECT clause. As an example, consider the following query:

```
SELECT country, YEAR(hiredate) AS yearhired
FROM HR.Employees
WHERE yearhired >= 2014;
```

This query fails with the following error:

```
Msg 207, Level 16, State 1, Line 114
Invalid column name 'yearhired'.
```

If you understand that the WHERE clause is evaluated before the SELECT clause, you realize that this attempt is wrong because at this phase, the attribute yearhired doesn't yet exist. You can indicate the expression YEAR(hiredate) >= 2014 in the WHERE clause. Better yet, for optimization reasons that are discussed later in Skill 1.3 in the section "Search arguments," use the form hiredate >= '20140101' as done in the original query.

3. GROUP ROWS BASED ON THE GROUP BY CLAUSE

This phase defines a group for each distinct combination of values in the grouped elements from the input virtual table. It then associates each input row to its respective group. The query you've been working with groups the rows by country and YEAR(hiredate). Within the six rows in the input table, this step identifies four groups. Here are the groups and the detail rows that are associated with them (redundant information removed for purposes of illustration).

```
group    group           detail detail   detail
country  year(hiredate)  empid  country  hiredate
-------- --------------- ------ -------- ----------
UK       2014            5      UK       2014-10-17
                         6      UK       2014-10-17
UK       2015            7      UK       2015-01-02
                         9      UK       2015-11-15
USA      2014            4      USA      2014-05-03
USA      2015            8      USA      2015-03-05
```

As you can see, the group UK, 2014 has two associated detail rows with employees 5 and 6; the group for UK, 2015 also has two associated detail rows with employees 7 and 9; the group for USA, 2014 has one associated detail row with employee 4; the group for USA, 2015 also has one associated detail row with employee 8.

The final result of this query has one row representing each group (unless filtered out). Therefore, expressions in all phases that take place after the current grouping phase are somewhat limited. All expressions processed in subsequent phases must guarantee a single value per group. If you refer to an element from the GROUP BY list (for example, country), you already have such a guarantee, so such a reference is allowed. However, if you want to refer to an element that is not part of your GROUP BY list (for example, empid), it must be contained within an aggregate function like MAX. That's because multiple values are possible in the element within a single group, and the only way to guarantee that just one will be returned is to aggregate the values.

4. FILTER GROUPS BASED ON THE HAVING CLAUSE

This phase is also responsible for filtering data based on a predicate, but it is evaluated after the data has been grouped; hence, it is evaluated per group and filters groups as a whole. As is usual in T-SQL, the filtering predicate can evaluate to true, false, or unknown. Only groups for which the predicate evaluates to true are returned from this phase. In this case, the HAVING clause uses the predicate COUNT(*) > 1, meaning filter only country and hire year groups that have more than one employee. If you look at the number of rows that were associated with each group in the previous step, you will notice that only the groups UK, 2014 and UK, 2015 qualify. Hence, the result of this phase has the following remaining groups, shown here with their associated detail rows.

```
group    group           detail detail   detail
country  year(hiredate)  empid  country  hiredate
-------- --------------- ------ -------- ----------
UK       2014            5      UK       2014-10-17
                         6      UK       2014-10-17
UK       2015            7      UK       2015-01-02
                         9      UK       2015-11-15
```

It's important to understand the difference between WHERE and HAVING. The WHERE clause is evaluated before rows are grouped, and therefore is evaluated per row. The HAVING clause is evaluated after rows are grouped, and therefore is evaluated per group.

5. PROCESS THE SELECT CLAUSE

The fifth phase is the one responsible for processing the SELECT clause. What's interesting about it is the point in logical query processing where it gets evaluated—almost last. That's interesting considering the fact that the SELECT clause appears first in the query.

This phase includes two main steps. The first step is evaluating the expressions in the SELECT list and producing the result attributes. This includes assigning attributes with names if they are derived from expressions. Remember that if a query is a grouped query, each group is represented by a single row in the result. In the query, two groups remain after the processing of the HAVING filter. Therefore, this step generates two rows. In this case, the SELECT list returns for each country and order year group a row with the following attributes: country, YEAR(hiredate) aliased as yearhired, and COUNT(*) aliased as numemployees.

The second step in this phase is applicable if you indicate the DISTINCT clause, in which case this step removes duplicates. Remember that T-SQL is based on multiset theory more than it is on set theory, and therefore, if duplicates are possible in the result, it's your responsibility to remove those with the DISTINCT clause. In this query's case, this step is inapplicable. Here's the result of this phase in the query:

```
country         yearhired   numemployees
--------------- ----------- ------------
UK              2014        2
UK              2015        2
```

The fifth phase returns a relational result. Therefore, the order of the rows isn't guaranteed. In this query's case, there is an ORDER BY clause that guarantees the order in the result, but this will be discussed when the next phase is described. What's important to note is that the outcome of the phase that processes the SELECT clause is still relational.

Also, remember that this phase assigns column aliases, like yearhired and numemployees. This means that newly created column aliases are not visible to clauses processed in previous phases, like FROM, WHERE, GROUP BY, and HAVING.

Note that an alias created by the SELECT phase isn't even visible to other expressions that appear in the same SELECT list. For example, the following query isn't valid:

```
SELECT empid, country, YEAR(hiredate) AS yearhired, yearhired - 1 AS prevyear
FROM HR.Employees;
```

This query generates the following error:

```
Msg 207, Level 16, State 1, Line 117
Invalid column name 'yearhired'.
```

The reason that this isn't allowed is that all expressions that appear in the same logical query-processing step are treated as a set, and a set has no order. In other words, conceptually, T-SQL evaluates all expressions that appear in the same phase in an all-at-once manner. Note the use of the word conceptually. SQL Server won't necessarily physically process all expressions at the same point in time, but it has to produce a result as if it did. This behavior is different than many other programming languages where expressions usually get evaluated

in a left-to-right order, making a result produced in one expression visible to the one that appears to its right. But T-SQL is different.

6. HANDLE PRESENTATION ORDERING

The sixth phase is applicable if the query has an ORDER BY clause. This phase is responsible for returning the result in a specific presentation order according to the expressions that appear in the ORDER BY list. The query indicates that the result rows should be ordered first by country (in ascending order by default), and then by yearhired, descending, yielding the following output:

```
country         yearhired   numemployees
--------------- ----------- ------------
UK              2015        2
UK              2014        2
```

Notice that the ORDER BY clause is the first and only clause that is allowed to refer to column aliases defined in the SELECT clause. That's because the ORDER BY clause is the only one to be evaluated after the SELECT clause.

Unlike in previous phases where the result was relational, the output of this phase isn't relational because it has a guaranteed order. The result of this phase is what standard SQL calls a cursor. Note that the use of the term cursor here is conceptual. T-SQL also supports an object called a cursor that is defined based on a result of a query, and that allows fetching rows one at a time in a specified order.

You might care about returning the result of a query in a specific order for presentation purposes or if the caller needs to consume the result in that manner through some cursor mechanism that fetches the rows one at a time. But remember that such processing isn't relational. If you need to process the query result in a relational manner—for example, define a table expression like a view based on the query—the result will need to be relational. Also, sorting data can add cost to the query processing. If you don't care about the order in which the result rows are returned, you can avoid this unnecessary cost by not adding an ORDER BY clause.

A query can specify the TOP or OFFSET-FETCH filtering options. If it does, the same ORDER BY clause that is normally used to define presentation ordering also defines which rows to filter for these options. It's important to note that such a filter is processed after the SELECT phase evaluates all expressions and removes duplicates (in case a DISTINCT clause was specified). You might even consider the TOP and OFFSET-FETCH filters as being processed in their own phase number 7. The query doesn't indicate such a filter, and therefore, this phase is inapplicable in this case.

For more information about logical query processing, see the following article series on the topic in SQL Server Pro magazine:

- Part 1 at *http://sqlmag.com/sql-server/logical-query-processing-what-it-and-what-it-means-you*
- Part 2 at *http://sqlmag.com/sql-server/logical-query-processing-clause-and-joins*

- Part 3 at *http://sqlmag.com/sql-server/logical-query-processing-clause-and-apply*
- Part 4: *http://sqlmag.com/sql-server/logical-query-processing-clause-and-pivot*
- Part 5 at *http://sqlmag.com/sql-server/logical-query-processing-part-5-clause-and-unpivot*
- Part 6 at *http://sqlmag.com/sql-server-2016/logical-query-processing-part-6-where-clause*
- Part 7 at *http://sqlmag.com/sql-server/logical-query-processing-part-7-group-and-having*
- Part 8 at *http://sqlmag.com/sql-server/logical-query-processing-part-8-select-and-order*

Getting started with the SELECT statement

The FROM and SELECT clauses are two principal clauses that appear in almost every query that retrieves data. This section explains the purpose of these clauses, how to use them, and best practices associated with them. It also explains what regular and irregular identifiers are, and how to delimit identifiers.

The FROM clause

According to logical query processing, the FROM clause is the first clause to be evaluated logically in a SELECT query. The FROM clause has two main roles:

- It's the clause where you indicate the tables that you want to query.
- It's the clause where you can apply table operators like joins to input tables.

This section focuses on the first role.

As a basic example, assuming you are connected to the sample database TSQLV4, the following query uses the FROM clause to specify that HR.Employees is the table being queried:

```
SELECT empid, firstname, lastname, country
FROM HR.Employees;
```

Observe the use of the two-part name to refer to the table. The first part (HR) is the schema name and the second part (Employees) is the table name. In some cases, T-SQL supports omitting the schema name, as in FROM Employees, in which case it uses an implicit schema name resolution process. It is considered a best practice to always explicitly indicate the schema name. This practice can prevent you from ending up with a schema name that you did not intend to use, and can also remove the cost involved in the implicit resolution process, although this cost is minor.

In the FROM clause, you can alias the queried tables with your chosen names. You can use the form <table> <alias>, as in HR.Employees E, or <table> AS <alias>, as in HR.Employees AS E. The latter form is more readable. When using aliases, the convention is to use short names, typically one letter that is somehow indicative of the queried table, like E for Employees. Then, when referring to an ambiguous column name in a multi-table query (same column

name appears in multiple queried tables), to avoid ambiguity, you add the table alias as a column prefix.

Note that if you assign an alias to a table, you basically rename the table for the duration of the query. The original table name isn't visible anymore; only the alias is. Normally, you can prefix a column name you refer to in a query with the table name, as in Employees.empid. However, if you aliased the Employees table as E, the reference Employees.empid is invalid; you have to use E.empid, as the following example demonstrates:

```
SELECT E.empid, firstname, lastname, country
FROM HR.Employees AS E;
```

If you try running this code by using the full table name as the column prefix, the code will fail.

The SELECT clause

The SELECT clause of a query has two main roles:

- It evaluates expressions that define the attributes in the query's result, assigning them with aliases if needed.
- Using a DISTINCT clause, you can eliminate duplicate rows in the result if needed.

Let's start with the first role. Take the following query as an example:

```
SELECT empid, firstname, lastname
FROM HR.Employees;
```

The FROM clause indicates that the HR.Employees table is the input table of the query. The SELECT clause then projects only three of the attributes from the input as the returned attributes in the result of the query.

T-SQL supports using an asterisk (*) as an alternative to listing all attributes from the input tables, but this is considered a bad practice for a number of reasons. Often, you need to return only a subset of the input attributes, and using an * is just a matter of laziness. By returning more attributes than you really need, you can prevent SQL Server from using what would normally be considered covering indexes with respect to the interesting set of attributes. You also send more data than is needed over the network, and this can have a negative impact on the system's performance. In addition, the underlying table definition could change over time; even if, when the query was initially authored, * really represented all attributes you needed; it might not be the case anymore at a later point in time. For these reasons and others, it is considered a best practice to always explicitly list the attributes that you need.

In the SELECT clause, you can assign your own aliases to the expressions that define the result attributes. There are a number of supported forms of aliasing: <expression> AS <alias> as in empid AS employeeid, <expression> <alias> as in empid employeeid, and <alias> = <expression> as in employeeid = empid.

NOTE **COLUMN ALIASES**

The first form with the AS clause is recommended to use because it's both standard and is the most readable. The second form is both less readable and makes it hard to spot a bug involving a missing comma.

Consider the following query:

```
SELECT empid, firstname lastname
FROM HR.Employees;
```

The developer who authored the query intended to return the attributes empid, firstname, and lastname but missed indicating the comma between firstname and lastname. The query doesn't fail; instead, it returns the following result:

```
empid       lastname
----------- ----------
1           Sara
2           Don
3           Judy
...
```

Although not the author's intention, SQL Server interprets the request as assigning the alias lastname to the attribute firstname instead of returning both. If you're used to aliasing expressions with the space form as a common practice, it will be harder for you to spot such bugs.

There are two main uses for intentional attribute aliasing. One is renaming—when you need the result attribute to be named differently than the source attribute—for example, if you need to name the result attribute employeeid instead of empid, as follows:

```
SELECT empid AS employeeid, firstname, lastname
FROM HR.Employees;
```

Another use is to assign a name to an attribute that results from an expression that would otherwise be unnamed. For example, suppose you need to generate a result attribute from an expression that concatenates the firstname attribute, a space, and the lastname attribute. You use the following query:

```
SELECT empid, firstname + N' ' + lastname
FROM HR.Employees;
```

You get a nonrelational result because the result attribute has no name:

```
empid
----------- ------------------------------
1           Sara Davis
2           Don Funk
3           Judy Lew
...
```

By aliasing the expression, you assign a name to the result attribute, making the result of the query relational, as follows.

```
SELECT empid, firstname + N' ' + lastname AS fullname
FROM HR.Employees;
```

Here's an abbreviated form of the result of this query:

```
empid       fulllname
----------- ------------------------------
1           Sara Davis
2           Don Funk
3           Judy Lew
...
```

Remember that if duplicates are possible in the result, T-SQL won't try to eliminate those unless instructed. A result with duplicates is considered nonrelational because relations—being sets—are not supposed to have duplicates. Therefore, if duplicates are possible in the result, and you want to eliminate them in order to return a relational result, you can do so by adding a DISTINCT clause, as in the following query:

```
SELECT DISTINCT country, region, city
FROM HR.Employees;
```

The HR.Employees table has nine rows but five distinct locations; hence, the output of this query has five rows:

```
country         region          city
--------------- --------------- ---------------
UK              NULL            London
USA             WA              Kirkland
USA             WA              Redmond
USA             WA              Seattle
USA             WA              Tacoma
```

There's an interesting difference between standard SQL and T-SQL in terms of minimal SELECT query requirements. According to standard SQL, a SELECT query must have at minimum FROM and SELECT clauses. Conversely, T-SQL supports a SELECT query with only a SELECT clause and without a FROM clause. Such a query is as if issued against an imaginary table that has only one row. For example, the following query is invalid according to standard SQL, but is valid according to T-SQL:

```
SELECT 10 AS col1, 'ABC' AS col2;
```

The output of this query is a single row with attributes resulting from the expressions with names assigned using the aliases:

```
col1        col2
----------- ----
10          ABC
```

Delimiting identifiers

When referring to identifiers of attributes, schemas, tables, and other objects, there are cases in which you are required to use delimiters versus cases in which the use of delimiters is optional. T-SQL supports both a standard form to delimit identifiers using double quotation marks, as in "Sales"."Orders", as well as a proprietary form using square brackets, as in [Sales].[Orders]. The latter is the more commonly used, and recommended, form in T-SQL.

When the identifier is *regular*, delimiting it is optional. In a regular identifier, the identifier complies with the rules for formatting identifiers. The rules say that the first character must be a letter defined by the Unicode Standard 3.2 (a-z, A-Z, and letters from other Unicode languages), underscore (_), at sign (@), or number sign (#). Subsequent characters can include letters, decimal numbers, at sign, dollar sign ($), number sign, or underscore. The identifier cannot be a reserved keyword in T-SQL, cannot have embedded spaces, and must not include supplementary characters.

An identifier that doesn't comply with these rules must be delimited. For example, an attribute called 2017 is considered an irregular identifier because it starts with a digit, and therefore must be delimited as "2017" or [2017]. A regular identifier such as y2017 can be referenced without delimiters simply as y2017, or optionally it can be delimited. You might prefer not to delimit regular identifiers because the delimiters tend to clutter the code.

Filtering data with predicates

Filtering data is one of the most fundamental aspects of T-SQL querying. Almost every query that you write involves some form of filtering. This section covers filtering data with predicates. Later sections in this skill cover filtering data with the TOP and OFFSET-FETCH options.

Predicates and three-valued-logic

Let's use the HR.Employees table to demonstrate a few filtering examples. Run the following code to show the contents of this table:

```
SELECT empid, firstname, lastname, country, region, city
FROM HR.Employees;
```

This query generates the following output:

```
empid firstname  lastname   country  region  city
----- ---------- ---------- -------- ------- ---------
1     Sara       Davis      USA      WA      Seattle
2     Don        Funk       USA      WA      Tacoma
3     Judy       Lew        USA      WA      Kirkland
4     Yael       Peled      USA      WA      Redmond
5     Sven       Mortensen  UK       NULL    London
6     Paul       Suurs      UK       NULL    London
7     Russell    King       UK       NULL    London
8     Maria      Cameron    USA      WA      Seattle
9     Patricia   Doyle      UK       NULL    London
```

Consider the following query, which filters only employees from the US:

```
SELECT empid, firstname, lastname, country, region, city
FROM HR.Employees
WHERE country = N'USA';
```

In case you're wondering why the literal 'USA' is preceded with the letter N as a prefix, that's to denote a Unicode character string literal, because the country column is of the data type NVARCHAR. Had the country column been of a regular character string data type, such as VARCHAR, the literal should have been just 'USA'.

When NULLs are not possible in the data that you're filtering, such as in the above example, T-SQL uses two-valued logic; namely, for any given row the predicate can evaluate to either true or false. The filter returns only the rows for which the predicate evaluates to true and discards the ones for which the predicate evaluates to false. Therefore, this query returns the following output:

```
empid firstname  lastname   country  region  city
----- ---------- ---------- -------- ------- ---------
1     Sara       Davis      USA      WA      Seattle
2     Don        Funk       USA      WA      Tacoma
3     Judy       Lew        USA      WA      Kirkland
4     Yael       Peled      USA      WA      Redmond
8     Maria      Cameron    USA      WA      Seattle
```

However, when NULLs are possible in the data, things get trickier. For instance, consider the following query:

```
SELECT empid, firstname, lastname, country, region, city
FROM HR.Employees
WHERE region = N'WA';
```

Here you're looking for only those employees who are from Washington (have WA in the region attribute). It's clear that the predicate evaluates to true for rows that have WA in the region attribute and that those rows are returned. It's also clear that the predicate would have evaluated to false had there been any rows with a present region other than WA, for example CA, and that those rows would have been discarded. However, remember that the predicate evaluates to unknown for rows that have a NULL in the region attribute, and that the WHERE clause discards such rows. This happens to be the desired behavior in our case because you know that when the region is NULL, it can't be Washington. However, remember that even when you use the inequality operator <> a comparison with a NULL yields unknown. For instance, suppose that you wanted to return only employees with a region other than Washington, and that you used the following query in attempt to achieve this:

```
SELECT empid, firstname, lastname, country, region, city
FROM HR.Employees
WHERE region <> N'WA';
```

The predicate evaluates to false for rows with WA in the region attribute and those rows are discarded. The predicate would have evaluated to true had there been rows with a present region other than WA, and those rows would have been returned. However, the

predicate evaluates to unknown for rows with NULL in the region attribute, and those rows get discarded, even though you know that if region is NULL, it cannot be Washington. This query returns an empty set because our sample data contains only rows with either WA or NULL in the region attribute:

```
empid firstname  lastname   country  region  city
----- ---------- ---------- -------- ------- ---------
```

This is an example where you need to intervene and add some logic to your query to also return the rows where the region attribute is NULL. Be careful though not to use an equality operator when looking for a NULL because remember that nothing is considered equal to a NULL—not even another NULL. The following query still returns an empty set:

```
SELECT empid, firstname, lastname, country, region, city
FROM HR.Employees
WHERE region <> N'WA'
   OR region = NULL;
```

T-SQL supports the IS NULL and IS NOT NULL operators to check if a NULL is or isn't present, respectively. Here's the solution query that correctly handles NULLs:

```
SELECT empid, firstname, lastname, country, region, city
FROM HR.Employees
WHERE region <> N'WA'
   OR region IS NULL;
```

This query generates the following output:

```
empid firstname  lastname   country  region  city
----- ---------- ---------- -------- ------- ---------
5     Sven       Mortensen  UK       NULL    London
6     Paul       Suurs      UK       NULL    London
7     Russell    King       UK       NULL    London
9     Patricia   Doyle      UK       NULL    London
```

Combining predicates

You can combine predicates in the WHERE clause by using the logical operators AND and OR. You can also negate predicates by using the NOT logical operator. This section starts by describing important aspects of negation and then discusses combining predicates.

Negation of true and false is straightforward—NOT true is false, and NOT false is true.

What can be surprising to some is what happens when you negate unknown—NOT unknown is still unknown. Recall from the previous section the query that returned all employees from Washington; the query used the predicate region = N'WA' in the WHERE clause. Suppose that you want to return the employees that are not from WA, and for this you use the predicate NOT region = N'WA'. It's clear that cases that return false from the positive predicate (say the region is NY) return true from the negated predicate. It's also clear that cases that return true from the positive predicate (say the region is WA) return false from the negated predicate. However, when the region is NULL, both the positive predicate and the

negated one return unknown and the row is discarded. So the right way for you to include NULL cases in the result—if that's what you know that you need to do—is to use the IS NULL operator, as in NOT region = N'WA' OR region IS NULL.

As for combining predicates, there are several interesting things to note. Some precedence rules determine the logical evaluation order of the different predicates. The NOT operator precedes AND and OR, and AND precedes OR. For example, suppose that the WHERE filter in your query had the following combination of predicates:

```
WHERE col1 = 'w' AND col2 = 'x' OR col3 = 'y' AND col4 = 'z'
```

Because AND precedes OR, you get the equivalent of the following:

```
WHERE (col1 = 'w' AND col2 = 'x') OR (col3 = 'y' AND col4 = 'z')
```

Trying to express the operators as pseudo functions, this combination of operators is equivalent to OR(AND(col1 = 'w', col2 = 'x'), AND(col3 = 'y', col4 = 'z')).

Because parentheses have the highest precedence among all operators, you can always use those to fully control the logical evaluation order that you need, as the following example shows:

```
WHERE col1 = 'w' AND (col2 = 'x' OR col3 = 'y') AND col4 = 'z'
```

Again, using pseudo functions, this combination of operators and use of parentheses is equivalent to AND(col1 = 'w', OR(col2 = 'x', col3 = 'y'), col4 = 'z').

Recall that all expressions that appear in the same logical query-processing phase—for example, the WHERE phase—are conceptually evaluated at the same point in time. For example, consider the following filter predicate:

```
WHERE propertytype = 'INT' AND CAST(propertyval AS INT) > 10
```

Suppose that the table being queried holds different property values. The propertytype column represents the type of the property (an INT, a DATE, and so on), and the propertyval column holds the value in a character string. When propertytype is 'INT', the value in propertyval is convertible to INT; otherwise, not necessarily.

Some assume that unless precedence rules dictate otherwise, predicates will be evaluated from left to right, and that short-circuiting will take place when possible. In other words, if the first predicate propertytype = 'INT' evaluates to false, SQL Server won't evaluate the second predicate CAST(propertyval AS INT) > 10 because the result is already known. Based on this assumption, the expectation is that the query should never fail trying to convert something that isn't convertible.

The reality, though, is different. SQL Server does internally support a short-circuit concept; however, due to the all-at-once concept in the language, it is not necessarily going to evaluate the expressions in left-to-right order. It could decide, based on cost-related reasons, to start with the second expression, and then if the second expression evaluates to true, to evaluate the first expression as well. This means that if there are rows in the table where

propertytype is different than 'INT', and in those rows propertyval isn't convertible to INT, the query can fail due to a conversion error.

You can deal with this problem in a number of ways. A simple option is to use the TRY_CAST function instead of CAST. When the input expression isn't convertible to the target type, TRY_CAST returns a NULL instead of failing. And comparing a NULL to anything yields unknown. Eventually, you will get the correct result, without allowing the query to fail. So your WHERE clause should be revised as follows:

```
WHERE propertytype = 'INT' AND TRY_CAST(propertyval AS INT) > 10
```

Filtering character data

In many respects, filtering character data is the same as filtering other types of data. This section covers a couple of items that are specific to character data: proper form of literals and the LIKE predicate.

A literal has a type. If you write an expression that involves operands of different types, SQL Server will have to apply implicit conversion to align the types. Depending on the circumstances, implicit conversions can sometimes hurt performance. It is important to know the proper form of literals of different types and make sure you use the right ones. A classic example for using incorrect literal types is with Unicode character strings (NVARCHAR and NCHAR types). The right form for a Unicode character string literal is to prefix the literal with a capital N and delimit the literal with single quotation marks, for example, N'literal'. For a regular character string literal, you just delimit the literal with single quotation marks, for example, 'literal'. It's a typical bad habit to specify a regular character string literal when the filtered column is of a Unicode type, as in the following example:

```
SELECT empid, firstname, lastname
FROM HR.Employees
WHERE lastname = 'Davis';
```

Because the column and the literal have different types, SQL Server implicitly converts one operand's type to the other. In this example, fortunately, SQL Server converts the literal's type to the column's type, so it can still efficiently rely on indexing. However, there can be cases where implicit conversion hurts performance. It is a best practice to use the proper form as follows:

```
SELECT empid, firstname, lastname
FROM HR.Employees
WHERE lastname = N'Davis';
```

T-SQL provides the LIKE predicate, which you can use to filter character string data (regular and Unicode) based on pattern matching. The form of a predicate using LIKE is as follows:

```
<column> LIKE <pattern>
```

The LIKE predicate supports wildcards that you can use in your patterns. Table 1-1 describes the available wildcards, their meaning, and an example demonstrating their use.

TABLE 1-1 Wildcards used in LIKE patterns

Wildcard	Meaning	Example
% (percent sign)	Any string including an empty one	'D%': string starting with D
_ (underscore)	A single character	'_D%': string where second character is D
[<character list>]	A single character from a list	'[AC]%': string where first character is A or C
[<character range>]	A single character from a range	'[0-9]%': string where first character is a digit
[^<character list or range>]	A single character that is not in the list or range	'[^0-9]%': string where first character is not a digit

As an example, suppose you want to return all employees whose last name starts with the letter D. You would use the following query:

```
SELECT empid, firstname, lastname
FROM HR.Employees
WHERE lastname LIKE N'D%';
```

This query returns the following output:

```
empid       firstname  lastname
----------- ---------- --------------------
1           Sara       Davis
9           Patricia   Doyle
```

If you want to look for a character that is considered a wildcard, you can indicate it after a character that you designate as an escape character by using the ESCAPE keyword. For example, the expression col1 LIKE '!_%' ESCAPE '!' looks for strings that start with an underscore (_) by using an exclamation point (!) as the escape character. Alternatively, you can place the wildcard in square brackets, as in col1 LIKE '[_]%'.

Filtering date and time data

There are several important considerations when filtering date and time data. You want to think of things like how to express literals and how to filter ranges. Suppose that you need to query the Sales.Orders table and return only orders placed on February 12, 2016. You use the following query:

```
SELECT orderid, orderdate, empid, custid
FROM Sales.Orders
WHERE orderdate = '02/12/16';
```

If you're an American, this form probably means February 12, 2016, to you. However, if you're British, this form probably means December 2, 2016. If you're Japanese, it probably means December 16, 2002. The question is, when SQL Server converts this character string to a date and time type to align it with the filtered column's type, how does it interpret the value? As it turns out, it depends on the language of the login that runs the code. Each login has a default language associated with it, and the default language sets various session options

on the login's behalf, including one called DATEFORMAT. A login with us_english will have the DATEFORMAT setting set to mdy, British to dmy, and Japanese to ymd. The problem is, how do you as a developer express a date if you want it to be interpreted the way you intended, regardless of who runs your code?

There are two main approaches. One is to use a form that is considered language-neutral. For example, the form '20160212' is always interpreted as ymd, regardless of your language. Note that the form '2016-02-12' is considered language-neutral only for the data types DATE, DATETIME2, and DATETIMEOFFSET. Unfortunately, due to historic reasons, this form is considered language-dependent for the types DATETIME and SMALLDATETIME. The advantage of the form without the separators is that it is language-neutral for all date and time types. So the recommendation is to write the query as follows:

```
SELECT orderid, orderdate, empid, custid
FROM Sales.Orders
WHERE orderdate = '20160212';
```

Another approach is to explicitly convert the string to the target type using the CONVERT function, and indicating the style number that represents the style that you used. You can find the documentation of the CONVERT function with the different style numbers that it supports at *https://msdn.microsoft.com/en-GB/library/ms187928.aspx*. For instance, to use the U.S. style, specify style number 101, as CONVERT(DATE, '02/12/2016', 101).

When filtering data stored in a DATETIME data type, you need to be very careful with ranges. To demonstrate why, first run the following code to create a table called Sales.Orders2 and populate it with a copy of the data from Sales.Orders, using the DATETIME data type for the orderdate attribute:

```
DROP TABLE IF EXISTS Sales.Orders2;

SELECT orderid, CAST(orderdate AS DATETIME) AS orderdate, empid, custid
INTO Sales.Orders2
FROM Sales.Orders;
```

Suppose you need to filter only the orders that were placed in April 2016. You use the following query in attempt to achieve this, thinking that the DATETIME type supports a three-digit precision as the fraction of the second:

```
SELECT orderid, orderdate, empid, custid
FROM Sales.Orders2
WHERE orderdate BETWEEN '20160401' AND '20160430 23:59:59.999';
```

In practice, though, the precision of DATETIME is three and a third milliseconds. Because 999 is not a multiplication of this precision, the value is rounded up to the next millisecond, which happens to be the midnight of the next day. To make matters worse, when people want to store only a date in a DATETIME type, they store the date with midnight in the time, so besides returning orders placed in April 2016, this query also returns all orders placed in May 1, 2016. Here's the output of this query, shown here in abbreviated format:

```
orderid     orderdate               empid       custid
----------- ----------------------- ----------- -----------
10990       2016-04-01 00:00:00.000 2           20
...
11063       2016-04-30 00:00:00.000 3           37
11064       2016-05-01 00:00:00.000 1           71
11065       2016-05-01 00:00:00.000 8           46
11066       2016-05-01 00:00:00.000 7           89

(77 row(s) affected)
```

The recommended way to express a date and time range is with a closed-open interval as follows:

```
SELECT orderid, orderdate, empid, custid
FROM Sales.Orders2
WHERE orderdate >= '20160401' AND orderdate < '20160501';
```

This time the output contains only the orders placed in April 2016.

Sorting data

One of the most confusing aspects of working with T-SQL is understanding when a query result is guaranteed to be ordered versus when it isn't. Correct understanding of this aspect of the language ties directly to the foundations of T-SQL—particularly mathematical set theory. If you understand this from the very early stages of writing T-SQL code, you will have a much easier time than many who simply have incorrect assumptions and expectations from the language.

Consider the following query as an example:

```
SELECT empid, firstname, lastname, city, MONTH(birthdate) AS birthmonth
FROM HR.Employees
WHERE country = N'USA' AND region = N'WA';
```

Is there a guarantee that the rows will be returned in particular order, and if so, what is that order? Some make an intuitive assumption that the rows will be returned in insertion order; some assume primary key order; some assume clustered index order; others know that there's no guarantee for any kind of order.

Remember that a table in T-SQL is supposed to represent a relation; a relation is a set, and a set has no order to its elements. With this in mind, unless you explicitly instruct the query otherwise, the result of a query has no guaranteed order. For example, this query gave the following output when run on one system:

```
empid       firstname  lastname            city            birthmonth
----------- ---------- ------------------- --------------- -----------
1           Sara       Davis               Seattle         12
2           Don        Funk                Tacoma          2
3           Judy       Lew                 Kirkland        8
4           Yael       Peled               Redmond         9
8           Maria      Cameron             Seattle         1
```

It might seem like the output is sorted by empid, but that's not guaranteed. What could be more confusing is that if you run the query repeatedly, it seems like the result keeps being returned in the same order; but again, that's not guaranteed. When the database engine (SQL Server in this case) processes this query, it knows that it can return the data in any order because there is no explicit instruction to return the data in a specific order. It could be that, due to optimization and other reasons, the SQL Server database engine chose to process the data in a particular way this time. There's even some likelihood that such choices will be repeated if the physical circumstances remain the same. But there's a big difference between what's likely to happen due to optimization and other reasons, and what's actually guaranteed.

The database engine can—and sometimes does—change choices that can affect the order in which rows are returned, knowing that it is free to do so. Examples for such changes in choices include changes in data distribution, availability of physical structures such as indexes, and availability of resources like CPU and memory. Also, with changes in the engine after an upgrade to a newer version of the product, or even after application of a service pack, optimization aspects can change. In turn, such changes could affect, among other things, the order of the rows in the result.

In short, this cannot be stressed enough: A query that doesn't have an explicit instruction to return the rows in a particular order doesn't guarantee the order of rows in the result. When you do need such a guarantee, the only way to provide it is by adding an ORDER BY clause to the query.

For example, if you want to return information about employees from Washington in the US, sorted by city, you specify the city column in the ORDER BY clause as follows:

```
SELECT empid, firstname, lastname, city, MONTH(birthdate) AS birthmonth
FROM HR.Employees
WHERE country = N'USA' AND region = N'WA'
ORDER BY city;
```

Here's the output of this query:

```
empid       firstname  lastname             city            birthmonth
----------- ---------- -------------------- --------------- -----------
3           Judy       Lew                  Kirkland        8
4           Yael       Peled                Redmond         9
8           Maria      Cameron              Seattle         1
1           Sara       Davis                Seattle         12
2           Don        Funk                 Tacoma          2
```

If you don't indicate a direction for sorting, ascending order is assumed by default. You can be explicit and specify city ASC, but it means the same thing as not indicating the direction. For descending ordering, you need to explicitly specify DESC, as follows:

```
SELECT empid, firstname, lastname, city, MONTH(birthdate) AS birthmonth
FROM HR.Employees
WHERE country = N'USA' AND region = N'WA'
ORDER BY city DESC;
```

This time, the output shows the rows in city order, descending direction:

```
empid       firstname  lastname            city            birthmonth
----------- ---------- ------------------- --------------- -----------
2           Don        Funk                Tacoma          2
1           Sara       Davis               Seattle         12
8           Maria      Cameron             Seattle         1
4           Yael       Peled               Redmond         9
3           Judy       Lew                 Kirkland        8
```

The city column isn't unique within the filtered country and region, and therefore, the ordering of rows with the same city (see Seattle, for example) isn't guaranteed. In such a case, it is said that the ordering isn't deterministic. Just like a query without an ORDER BY clause doesn't guarantee order among result rows in general, a query with ORDER BY city, when city isn't unique, doesn't guarantee order among rows with the same city. Fortunately, you can specify multiple expressions in the ORDER BY list, separated by commas. One use case of this capability is to apply a tiebreaker for ordering. For example, you could define empid as the secondary sort column, as follows:

```
SELECT empid, firstname, lastname, city, MONTH(birthdate) AS birthmonth
FROM HR.Employees
WHERE country = N'USA' AND region = N'WA'
ORDER BY city, empid;
```

Here's the output of this query:

```
empid       firstname  lastname            city            birthmonth
----------- ---------- ------------------- --------------- -----------
3           Judy       Lew                 Kirkland        8
4           Yael       Peled               Redmond         9
1           Sara       Davis               Seattle         12
8           Maria      Cameron             Seattle         1
2           Don        Funk                Tacoma          2
```

The ORDER BY list is now unique; hence, the ordering is deterministic. As long as the underlying data doesn't change, the results are guaranteed to be repeatable, in addition to their presentation ordering. You can indicate the ordering direction on an expression-by-expression basis, as in ORDER BY col1 DESC, col2, col3 DESC (col1 descending, then col2 ascending, then col3 descending).

With T-SQL, you can sort by ordinal positions of columns in the SELECT list, but it is considered a bad practice. Consider the following query as an example:

```
SELECT empid, firstname, lastname, city, MONTH(birthdate) AS birthmonth
FROM HR.Employees
WHERE country = N'USA' AND region = N'WA'
ORDER BY 4, 1;
```

In this query, you're asking to order the rows by the fourth expression in the SELECT list (city), and then by the first (empid). In this particular query, it is equivalent to using ORDER BY city, empid. However, this practice is considered a bad one for a number of reasons. For one, T-SQL does keep track of ordinal positions of columns in a table, in addition to in a query result, but this is nonrelational. Recall that the heading of a relation is a set of attributes, and

a set has no order. Also, when you are using ordinal positions, it is very easy after making changes to the SELECT list to miss changing the ordinals accordingly. For example, suppose that you decide to apply changes to your previous query, returning city right after empid in the SELECT list. You apply the change to the SELECT list but forget to change the ORDER BY list accordingly, and end up with the following query:

```
SELECT empid, city, firstname, lastname, MONTH(birthdate) AS birthmonth
FROM HR.Employees
WHERE country = N'USA' AND region = N'WA'
ORDER BY 4, 1;
```

Now the query is ordering the data by lastname and empid instead of by city and empid. In short, it's a best practice to refer to column names, or expressions based on column names, and not to ordinal positions.

Note that you can order the result rows by elements that you're not returning. For example, the following query returns, for each qualifying employee, the employee ID and city, ordering the result rows by the employee birth date:

```
SELECT empid, city
FROM HR.Employees
WHERE country = N'USA' AND region = N'WA'
ORDER BY birthdate;
```

Here's the output of this query:

```
empid       city
----------- ---------------
4           Redmond
1           Seattle
2           Tacoma
8           Seattle
3           Kirkland
```

Of course, the result would appear much more meaningful if you included the birthdate attribute, but if it makes sense for you not to, it's perfectly valid. The rule is that you can order the result rows by elements that are not part of the SELECT list, as long as those elements would have normally been allowed there. This rule changes when the DISTINCT clause is also specified, and for a good reason. When DISTINCT is used, duplicates are removed; then the result rows don't necessarily map to source rows in a one-to-one manner, rather than one-to-many. For example, try to reason why the following query isn't valid:

```
SELECT DISTINCT city
FROM HR.Employees
WHERE country = N'USA' AND region = N'WA'
ORDER BY birthdate;
```

You can have multiple employees—each with a different birth date—from the same city. But you're returning only one row for each distinct city in the result. So given one city (say, Seattle) with multiple employees, which of the employee birth dates should apply as the ordering value? The query won't just pick one; rather, it simply fails.

So, in case the DISTINCT clause is used, you are limited in the ORDER BY list to only elements that appear in the SELECT list, as in the following query:

```
SELECT DISTINCT city
FROM HR.Employees
WHERE country = N'USA' AND region = N'WA'
ORDER BY city;
```

Now the query is perfectly sensible, returning the following output:

```
city
---------------
Kirkland
Redmond
Seattle
Tacoma
```

What's also interesting to note about the ORDER BY clause is that it gets evaluated conceptually after the SELECT clause—unlike most other query clauses. This means that column aliases assigned in the SELECT clause are actually visible to the ORDER BY clause. As an example, the following query uses the MONTH function to return the birth month, assigning the expression with the column alias birthmonth. The query then refers to the column alias birthmonth directly in the ORDER BY clause:

```
SELECT empid, firstname, lastname, city, MONTH(birthdate) AS birthmonth
FROM HR.Employees
WHERE country = N'USA' AND region = N'WA'
ORDER BY birthmonth;
```

This query returns the following output:

```
empid       firstname  lastname            city            birthmonth
----------- ---------- -------------------- --------------- -----------
8           Maria      Cameron              Seattle         1
2           Don        Funk                 Tacoma          2
3           Judy       Lew                  Kirkland        8
4           Yael       Peled                Redmond         9
1           Sara       Davis                Seattle         12
```

Another tricky aspect of ordering is treatment of NULLs. Recall that a NULL represents a missing value, so when comparing a NULL to anything, you get the logical result unknown. That's the case even when comparing two NULLs. So it's not that trivial to ask how NULLs should behave in terms of sorting. Should they all sort together? If so, should they sort before or after non-NULL values? Standard SQL says that NULLs should sort together, but leaves it to the implementation to decide whether to sort them before or after non-NULL values. In SQL Server the decision was to sort them before non-NULLs (when using an ascending direction). As an example, the following query returns for each order the order ID and shipped date, ordered by the latter:

```
SELECT orderid, shippeddate
FROM Sales.Orders
WHERE custid = 20
ORDER BY shippeddate;
```

Remember that unshipped orders have a NULL in the shippeddate column; hence, they sort before shipped orders, as the query output shows:

```
orderid     shippeddate
----------- -----------
11008       NULL
11072       NULL
10258       2014-07-23
10263       2014-07-31
…
```

Standard SQL supports the options NULLS FIRST and NULLS LAST to control how NULLs sort, but T-SQL doesn't support this option. As an interesting challenge, see if you can figure out how to sort the orders by shipped date ascending, but have NULLs sort last. (Hint: You can specify expressions in the ORDER BY clause; think of how to use the CASE expression to achieve this task.)

So remember, a query without an ORDER BY clause returns a relational result (at least from an ordering perspective), and hence doesn't guarantee any order. The only way to guarantee order is with an ORDER BY clause. According to standard SQL, a query with an ORDER BY clause conceptually returns a cursor and not a relation.

Creating the right indexes can help SQL Server avoid the need to actually sort the data to address an ORDER BY request. Without good indexes, SQL Server needs to sort the data, and sorting can be expensive, especially when a large set is involved. If you don't need to return the data sorted, make sure you do not specify an ORDER BY clause, to avoid unnecessary costs.

Filtering data with TOP and OFFSET-FETCH

Besides supporting filters that are based on predicates, like the WHERE filter, T-SQL also supports filters that are based on a number, or percent of rows and ordering. Those are the TOP and OFFSET-FETCH filters. The former is used in a lot of common filtering tasks, and the latter is typically used in more specialized paging-related tasks.

Filtering data with TOP

With the TOP option, you can filter a requested number or percent of rows from the query result based on indicated ordering. You specify the TOP option in the SELECT clause followed by the requested number of rows in parentheses (as a BIGINT typed value). The ordering specification of the TOP filter is based on the same ORDER BY clause that is normally used for presentation ordering.

As an example, the following query returns the three most recent orders:

```
SELECT TOP (3) orderid, orderdate, custid, empid
FROM Sales.Orders
ORDER BY orderdate DESC;
```

You specify 3 as the number of rows you want to filter, and orderdate DESC as the ordering specification. So you get the three rows with the most recent order dates. Here's the output of this query:

```
orderid     orderdate  custid      empid
----------- ---------- ----------- -----------
11077       2016-05-06 65          1
11076       2016-05-06 9           4
11075       2016-05-06 68          8
```

EXAM TIP

T-SQL supports specifying the number of rows to filter using the TOP option in SELECT queries without parentheses, but that's only for backward-compatibility reasons. The correct syntax is with parentheses.

You can also specify a percent of rows to filter instead of a number. To do so, specify a FLOAT value in the range 0 through 100 in the parentheses, and the keyword PERCENT after the parentheses, as follows:

```
SELECT TOP (1) PERCENT orderid, orderdate, custid, empid
FROM Sales.Orders
ORDER BY orderdate DESC;
```

The PERCENT option computes the ceiling of the resulting number of rows if it's not whole. In this example, without the TOP option, the number of rows in the result is 830. Filtering 1 percent gives you 8.3, and then the ceiling of this value gives you 9; hence, the query returns 9 rows:

```
orderid     orderdate  custid      empid
----------- ---------- ----------- -----------
11074       2016-05-06 73          7
11075       2016-05-06 68          8
11076       2016-05-06 9           4
11077       2016-05-06 65          1
11070       2016-05-05 44          2
11071       2016-05-05 46          1
11072       2016-05-05 20          4
11073       2016-05-05 58          2
11067       2016-05-04 17          1
```

The TOP option isn't limited to a constant input; instead, it allows you to specify a self-contained expression. From a practical perspective, this capability is especially important when you need to pass a parameter or a variable as input, as the following code demonstrates:

```
DECLARE @n AS BIGINT = 5;

SELECT TOP (@n) orderid, orderdate, custid, empid
FROM Sales.Orders
ORDER BY orderdate DESC;
```

This query generates the following output:

```
orderid     orderdate  custid      empid
----------- ---------- ----------- -----------
11074       2016-05-06 73          7
11075       2016-05-06 68          8
11076       2016-05-06 9           4
11077       2016-05-06 65          1
11070       2016-05-05 44          2
```

In most cases, you need your TOP option to rely on some ordering specification, but as it turns out, an ORDER BY clause isn't mandatory. For example, the following query is technically valid:

```
SELECT TOP (3) orderid, orderdate, custid, empid
FROM Sales.Orders;
```

However, the query isn't deterministic. The query filters three rows, but you have no guarantee which three rows will be returned. You end up getting whichever three rows SQL Server happened to access first, and that's dependent on physical data layout and optimization choices, none of which is guaranteed to be repeatable. For example, this query gave the following output on one system:

```
orderid     orderdate  custid      empid
----------- ---------- ----------- -----------
10248       2014-07-04 85          5
10249       2014-07-05 79          6
10250       2014-07-08 34          4
```

But there's no guarantee that the same rows will be returned if you run the query again. If you are really after three arbitrary rows, it might be a good idea to add an ORDER BY clause with the expression (SELECT NULL) to let people know that your choice is intentional and not an oversight. Here's how your query would look:

```
SELECT TOP (3) orderid, orderdate, custid, empid
FROM Sales.Orders
ORDER BY (SELECT NULL);
```

Note that even when you do have an ORDER BY clause, in order for the query to be completely deterministic, the ordering must be unique. For example, consider again the first query from this section:

```
SELECT TOP (3) orderid, orderdate, custid, empid
FROM Sales.Orders
ORDER BY orderdate DESC;
```

The orderdate column isn't unique, so the ordering in case of ties is arbitrary. When I ran this query on my system, I received the following output.

```
orderid     orderdate  custid      empid
----------- ---------- ----------- -----------
11077       2016-05-06 65          1
11076       2016-05-06 9           4
11075       2016-05-06 68          8
```

But what if there are other rows in the result without TOP that have the same order date as in the last row here? You don't always care about guaranteeing deterministic or repeatable results; but if you do, two options are available to you. One option is to ask to include all ties with the last row by adding the WITH TIES option, as follows:

```
SELECT TOP (3) WITH TIES orderid, orderdate, custid, empid
FROM Sales.Orders
ORDER BY orderdate DESC;
```

Of course, this could result in returning more rows than you asked for, as the output of this query shows:

```
orderid     orderdate  custid      empid
----------- ---------- ----------- -----------
11074       2016-05-06 73          7
11075       2016-05-06 68          8
11076       2016-05-06 9           4
11077       2016-05-06 65          1
```

Now the selection of rows is deterministic, but still the presentation order between rows with the same order date isn't.

The other option to guarantee determinism is to break the ties by adding a tiebreaker that makes the ordering unique. For example, in case of ties in the order date, suppose you wanted to use the order ID, descending, as the tiebreaker. To do so, add orderid DESC to your ORDER BY clause, as follows:

```
SELECT TOP (3) orderid, orderdate, custid, empid
FROM Sales.Orders
ORDER BY orderdate DESC, orderid DESC;
```

Now both the selection of rows and presentation order are deterministic. This query generates the following output:

```
orderid     orderdate  custid      empid
----------- ---------- ----------- -----------
11077       2016-05-06 65          1
11076       2016-05-06 9           4
11075       2016-05-06 68          8
```

Filtering data with OFFSET-FETCH

The OFFSET-FETCH option is a filtering option that, like TOP, you can use to filter data based on a specified number of rows and ordering. But unlike TOP, it is standard, and also has a skipping capability, making it useful for ad-hoc paging purposes.

The OFFSET and FETCH clauses appear right after the ORDER BY clause, and in fact, in T-SQL, they require an ORDER BY clause to be present. You first specify the OFFSET clause indicating how many rows you want to skip (0 if you don't want to skip any); you then optionally specify the FETCH clause indicating how many rows you want to filter. For example, the following query defines ordering based on order date descending, followed by order ID descending; it then skips the first 50 rows and fetches the next 25 rows:

```
SELECT orderid, orderdate, custid, empid
FROM Sales.Orders
ORDER BY orderdate DESC, orderid DESC
OFFSET 50 ROWS FETCH NEXT 25 ROWS ONLY;
```

Here's an abbreviated form of the output:

```
orderid     orderdate  custid      empid
----------- ---------- ----------- -----------
11027       2016-04-16 10          1
11026       2016-04-15 27          4
...
11004       2016-04-07 50          3
11003       2016-04-06 78          3
```

The ORDER BY clause now plays two roles: One role is telling the OFFSET-FETCH option which rows it needs to filter. Another role is determining presentation ordering in the query.

As mentioned, in T-SQL, the OFFSET-FETCH option requires an ORDER BY clause to be present. Also, in T-SQL—contrary to standard SQL—a FETCH clause requires an OFFSET clause to be present. So if you do want to filter some rows but skip none, you still need to specify the OFFSET clause with 0 ROWS.

In order to make the syntax intuitive, you can use the keywords NEXT or FIRST interchangeably. When skipping some rows, it might be more intuitive to you to use the keywords FETCH NEXT to indicate how many rows to filter; but when not skipping any rows, it might be more intuitive to you to use the keywords FETCH FIRST, as follows:

```
SELECT orderid, orderdate, custid, empid
FROM Sales.Orders
ORDER BY orderdate DESC, orderid DESC
OFFSET 0 ROWS FETCH FIRST 25 ROWS ONLY;
```

For similar reasons, you can use the singular form ROW or the plural form ROWS interchangeably, both for the number of rows to skip and for the number of rows to filter. But it's not like you will get an error if you say FETCH NEXT 1 ROWS or FETCH NEXT 25 ROW. It's up to you to use a proper form, just like with English.

In T-SQL, a FETCH clause requires an OFFSET clause, but the OFFSET clause doesn't require a FETCH clause. In other words, by indicating an OFFSET clause, you're requesting to skip some rows; then by not indicating a FETCH clause, you're requesting to return all remaining rows. For example, the following query requests to skip 50 rows, returning all the rest.

```
SELECT orderid, orderdate, custid, empid
FROM Sales.Orders
ORDER BY orderdate DESC, orderid DESC
OFFSET 50 ROWS;
```

This query generates the following output, shown here in abbreviated form:

```
orderid     orderdate  custid      empid
----------- ---------- ----------- -----------
11027       2016-04-16 10          1
11026       2016-04-15 27          4
...
10249       2014-07-05 79          6
10248       2014-07-04 85          5

(780 row(s) affected)
```

As mentioned earlier, the OFFSET-FETCH option requires an ORDER BY clause. But what if you need to filter a certain number of rows based on arbitrary order? To do so, you can specify the expression (SELECT NULL) in the ORDER BY clause, as follows:

```
SELECT orderid, orderdate, custid, empid
FROM Sales.Orders
ORDER BY (SELECT NULL)
OFFSET 0 ROWS FETCH FIRST 3 ROWS ONLY;
```

This code simply filters three arbitrary rows. Here's the output that I got when running this query on our system:

```
orderid     orderdate  custid      empid
----------- ---------- ----------- -----------
10248       2014-07-04 85          5
10249       2014-07-05 79          6
10250       2014-07-08 34          4
```

With both the OFFSET and the FETCH clauses, you can use expressions as inputs. This is very handy when you need to compute the input values dynamically. For example, suppose that you're implementing a paging solution where you return to the user one page of rows at a time. The user passes as input parameters to your procedure or function the page number they are after (@pagenum parameter) and page size (@pagesize parameter). This means that you need to skip as many rows as @pagenum minus one times @pagesize, and fetch the next @pagesize rows. This can be implemented using the following code (using local variables for simplicity):

```
DECLARE @pagesize AS BIGINT = 25, @pagenum AS BIGINT = 3;

SELECT orderid, orderdate, custid, empid
FROM Sales.Orders
ORDER BY orderdate DESC, orderid DESC
OFFSET (@pagenum - 1) * @pagesize ROWS FETCH NEXT @pagesize ROWS ONLY;
```

With these inputs, the code returns the following output:

```
orderid     orderdate  custid      empid
----------- ---------- ----------- -----------
11027       2016-04-16 10          1
11026       2016-04-15 27          4
...
11004       2016-04-07 50          3
11003       2016-04-06 78          3
```

You can feel free to change the input values and see how the result changes accordingly.

EXAM TIP

In terms of logical query processing, the TOP and OFFSET-FETCH filters are processed after the FROM, WHERE, GROUP, HAVING and SELECT phases. You can consider these filters as being an extension to the ORDER BY clause. So, for example, if the query is a grouped query, and also involves a TOP or OFFSET-FETCH filter, the filter is applied after grouping. The same applies if the query has a DISTINCT clause and/or ROW_NUMBER calculation as part of the SELECT clause, as well as a TOP or OFFSET-FETCH filter. The filter is applied after the DISTINCT clause and/or ROW_NUMBER calculation.

Because the OFFSET-FETCH option is standard and TOP isn't, in cases where they are logically equivalent, it's recommended to stick to the former. Remember that OFFSET-FETCH also has an advantage over TOP in the sense that it supports a skipping capability. However, for now, OFFSET-FETCH does not support options similar to TOP's PERCENT and WITH TIES, even though the standard does define them.

From a performance standpoint, you should consider indexing the ORDER BY columns to support the TOP and OFFSET-FETCH options. Such indexing serves a similar purpose to indexing filtered columns and can help avoid scanning unnecessary data as well as sorting.

MORE INFO **ON TOP AND OFFSET-FETCH**

For more information on the TOP and OFFSET-FETCH filters, see the free sample chapter of the book, "T-SQL Querying: Chapter 5 - TOP and OFFSET-FETCH." This book is more advanced, and includes detailed coverage of optimization aspects. You can find the online version of this chapter at *https://www.microsoftpressstore.com/articles/article.aspx?p=2314819*. You can download the PDF version of this chapter at *https://ptgmedia.pearsoncmg.com/images/9780735685048/samplepages/9780735685048.pdf*.

Combining sets with set operators

Set operators operate on two result sets of queries, comparing complete rows between the results. Depending on the result of the comparison and the operator used, the operator determines whether to return the row or not. T-SQL supports the operators: UNION, UNION ALL, INTERSECT, and EXCEPT.

The general form of code using these operators is as follows:

```
<query 1>
<operator>
<query 2>
[ORDER BY <order_by_list>];
```

When working with these operators you need to remember the following guidelines:

- Because complete rows are matched between the result sets, the number of columns in the queries has to be the same and the column types of corresponding columns need to be compatible (implicitly convertible).
- These operators use distinctness-based comparison and not equality based. Consequently, a comparison between two NULLs yields true, and a comparison between a NULL and a non-NULL value yields a false. This is in contrast to filtering clauses like WHERE, ON, and HAVING, which yield unknown when comparing a NULL with anything using both equality and inequality operators.
- Because these operators are set operators and not cursor operators, the individual queries are not allowed to have ORDER BY clauses.
- You can optionally add an ORDER BY clause that determines presentation ordering of the result of the set operator.
- The column names of result columns are determined by the first query.

EXAM TIP

The term *set operator* is not a precise term to describe the UNION, INERSECT, and EXCEPT operators, rather *relational operator* is a better term. Whereas in mathematical set theory you can unify a set of teachers with a set of prime numbers, in relational theory, you can't. You can only unify two relations that share the same attributes. This is explained in Dejan Sarka's blog post on the topic at *http://sqlblog.com/blogs/dejan_sarka/archive/2014/01/10/sql-set-operators-set-really.aspx*. However, both the SQL community and the official T-SQL documentation use the term set operator. Also, chances are that the same terminology will be used in the exam. Therefore, I am using this terminology in this book.

UNION and UNION ALL

The UNION operator unifies the results of the two input queries. As a set operator, UNION has an implied DISTINCT property, meaning that it does not return duplicate rows. Figure 1-2 shows an illustration of the UNION operator.

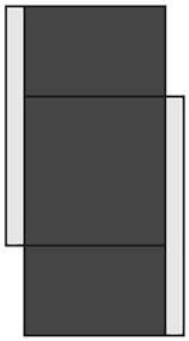


FIGURE 1-2 The UNION operator

As an example for using the UNION operator, the following query returns locations that are employee locations, or customer locations, or both:

```
SELECT country, region, city
FROM HR.Employees

UNION

SELECT country, region, city
FROM Sales.Customers;
```

This query generates the following output, shown here in abbreviated form:

```
country         region          city
--------------- --------------- ---------------
Argentina       NULL            Buenos Aires
Austria         NULL            Graz
Austria         NULL            Salzburg
...

(71 row(s) affected)
```

The HR.Employees table has nine rows, and the Sales.Customers table has 91 rows, but there are 71 distinct locations in the unified results; hence, the UNION operator returns 71 rows.

If you want to keep the duplicates—for example, to later group the rows and count occurrences—you need to use the UNION ALL operator instead of UNION. The UNION ALL operator unifies the results of the two input queries, but doesn't try to eliminate duplicates.

As an example, the following query unifies employee locations and customer locations using the UNION ALL operator:

```
SELECT country, region, city
FROM HR.Employees

UNION ALL

SELECT country, region, city
FROM Sales.Customers;
```

Because UNION ALL doesn't attempt to remove duplicates, the result has 100 rows (nine employees and 91 customers):

```
country         region          city
--------------- --------------- ---------------
USA             WA              Seattle
USA             WA              Tacoma
USA             WA              Kirkland
USA             WA              Redmond
UK              NULL            London
UK              NULL            London
UK              NULL            London
USA             WA              Seattle
UK              NULL            London
Germany         NULL            Berlin
...

(100 row(s) affected)
```

If the sets you're unifying are disjoint and there's no potential for duplicates, UNION and UNION ALL returns the same result. However, it's important to use UNION ALL in such a case from a performance standpoint because with UNION, SQL Server can try to eliminate duplicates, incurring unnecessary cost. Figure 1-3 shows the query execution plans for both the UNION (top plan) and UNION ALL (bottom) queries.

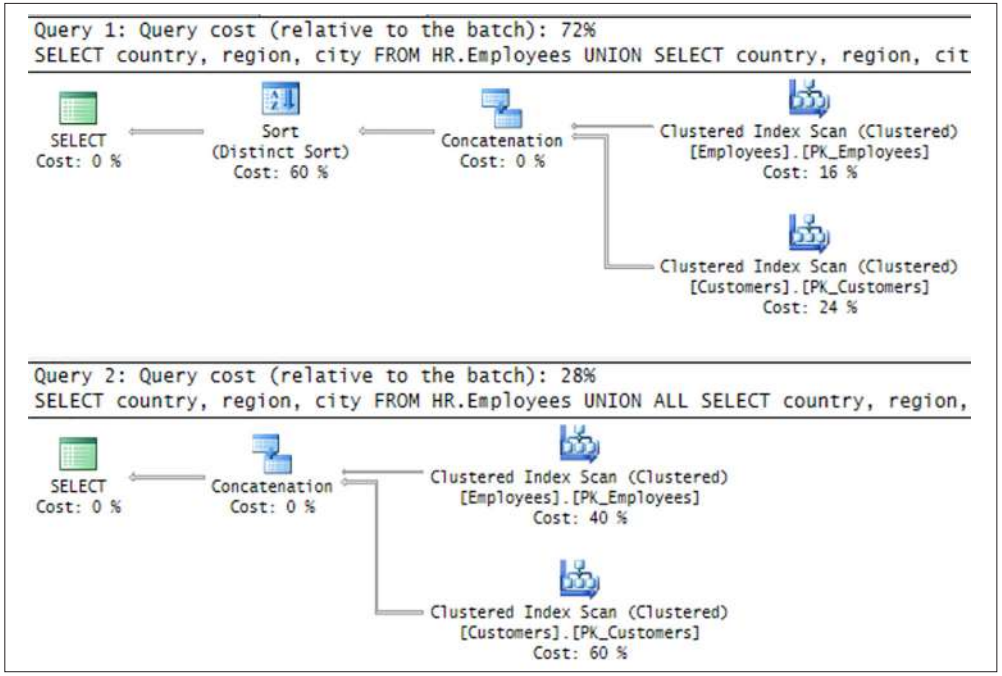


FIGURE 1-3 Query plan for UNION and UNION ALL operators

NOTE **DISPALYING AN ACTUAL EXECUTION PLAN**

To see graphical execution plans in SQL Server Management Studio (SSMS) click the Include Actual Execution Plan button, or press Ctrl+M, and run the code. The plans appear in the Execution Plan tab. You can find tips about analyzing query plans efficiently at *http://sqlmag.com/t-sql/understanding-query-plans*.

Observe that both plans start the same by scanning the two input tables and then concatenating (unifying) the results. But only the UNION operator includes an extra step with the Sort (Distinct Sort) operator to eliminate duplicates.

INTERSECT

The INTERSECT operator returns only distinct rows that are common to both sets. In other words, if a row appears at least once in the first set and at least once in the second set, it appears once in the result of the INTERSECT operator.

Figure 1-4 shows an illustration of the INTERSECT operator.

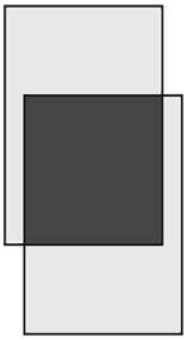


FIGURE 1-4 The INTERSECT operator

As an example, the following code uses the INTERSECT operator to return distinct locations that are both employee and customer locations (locations where there's at least one employee and at least one customer):

```
SELECT country, region, city
FROM HR.Employees

INTERSECT

SELECT country, region, city
FROM Sales.Customers;
```

This query generates the following output:

```
country         region          city
--------------- --------------- ---------------
UK              NULL            London
USA             WA              Kirkland
USA             WA              Seattle
```

Observe that the location (UK, NULL, London) was returned because it appears in both sides. When comparing the NULLs in the region column in the rows from the two sides, the INTERSECT operator considered them as not distinct from each other. Also note that never mind how many times the same location appears in each side, as long as it appears at least once in both sides, it's returned only once in the output.

EXCEPT

The EXCEPT operator performs set difference. It returns distinct rows that appear in the result of the first query but not the second. In other words, if a row appears at least once in the first query result and zero times in the second, it's returned once in the output.

Figure 1-5 shows an illustration of the EXCEPT operator.

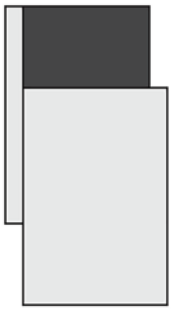


FIGURE 1-5 The EXCEPT operator

As an example for using EXCEPT, the following query returns locations that are employee locations but not customer locations:

```
SELECT country, region, city
FROM HR.Employees

EXCEPT

SELECT country, region, city
FROM Sales.Customers;
```

This query generates the following output:

```
country         region          city
--------------- --------------- ---------------
USA             WA              Redmond
USA             WA              Tacoma
```

With UNION and INTERSECT, the order of the input queries doesn't matter. However, with EXCEPT, there's different meaning to:

```
<query 1> EXCEPT <query 2>
```

Versus:

```
<query 2> EXCEPT <query 1>
```

Finally, set operators have precedence: INTERSECT precedes UNION and EXCEPT, and UNION and EXCEPT are evaluated from left to right based on their position in the expression. Consider the following set operators:

```
<query 1> UNION <query 2> INTERSECT <query 3>;
```

First, the intersection between query 2 and query 3 takes place, and then a union between the result of the intersection and query 1. You can always force precedence by using parentheses. So, if you want the union to take place first, you use the following form:

```
(<query 1> UNION <query 2>) INTERSECT <query 3>;
```

When you're done, run the following code for cleanup:

```
DROP TABLE IF EXISTS Sales.Orders2;
```

Skill 1.2: Query multiple tables by using joins

Often, data that you need to query is spread across multiple tables. The tables are usually related through keys, such as a foreign key in one side and a primary key in the other. Then you can use joins to query the data from the different tables and match the rows that need to be related. This section covers the different types of joins that T-SQL supports: cross, inner, and outer.

This section covers how to:

- Write queries with join statements based on provided tables, data, and requirements
- Determine proper usage of INNER JOIN, LEFT/RIGHT/FULL OUTER JOIN, and CROSS JOIN
- Construct multiple JOIN operators using AND and OR
- Determine the correct results when presented with multi-table SELECT statements and source data
- Write queries with NULLs on joins

Before running the code samples in this skill, add a row to the Suppliers table by running the following code:

```
USE TSQLV4;

INSERT INTO Production.Suppliers
  (companyname, contactname, contacttitle, address, city, postalcode, country, phone)
  VALUES(N'Supplier XYZ', N'Jiru', N'Head of Security', N'42 Sekimai Musashino-shi',
         N'Tokyo', N'01759', N'Japan', N'(02) 4311-2609');
```

Cross joins

A cross join is the simplest type of join, though not the most commonly used one. This join performs what's known as a Cartesian product of the two input tables. In other words, it performs a multiplication between the tables, yielding a row for each combination of rows from both sides. If you have m rows in table T1 and n rows in table T2, the result of a cross join between T1 and T2 is a virtual table with m × n rows. Figure 1-6 provides an illustration of a cross join.

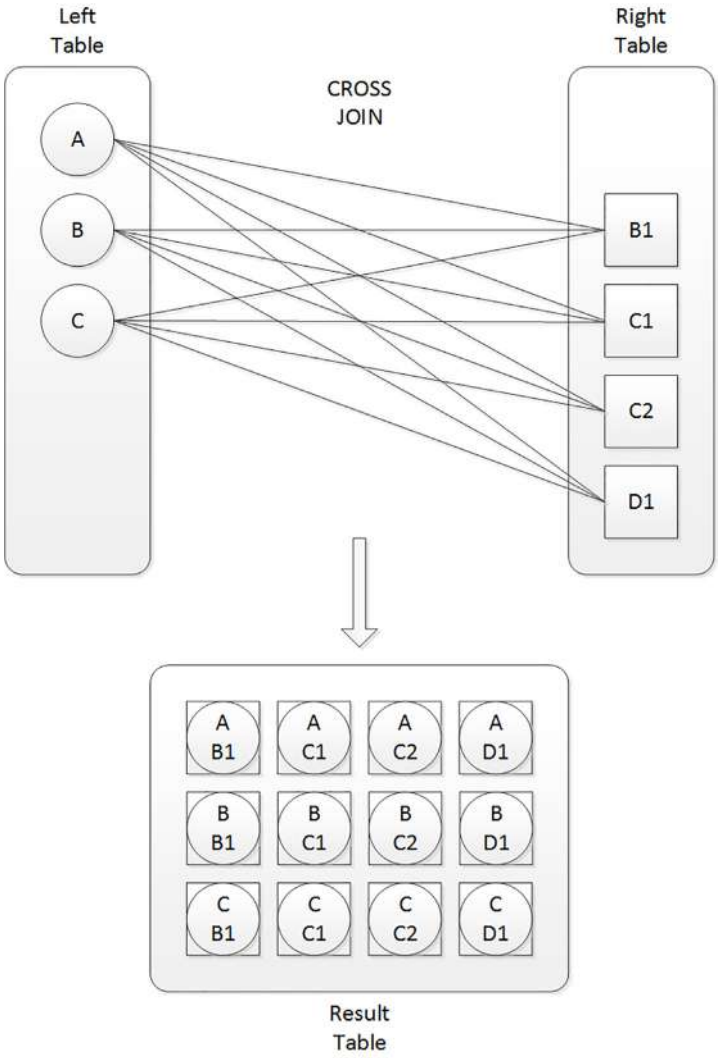


FIGURE 1-6 Cross join

The left table has three rows with the key values A, B, and C. The right table has four rows with the key values B1, C1, C2, and D1. The result is a table with 12 rows containing all possible combinations of rows from the two input tables.

Consider an example from the TSQLV4 sample database. This database contains a table called dbo.Nums that has a column called n with a sequence of integers from 1 and on. Your task is to use the Nums table to generate a result with a row for each weekday (1 through 7) and shift number (1 through 3), assuming there are three shifts a day. The result can later be used as the basis for building information about activities in the different shifts in the different days. With seven days in the week, and three shifts every day, the result should have 21 rows.

Here's a query that achieves the task by performing a cross join between two instances of the Nums table—one representing the days (aliased as D), and the other representing the shifts (aliased as S):

```
USE TSQLV4;

SELECT D.n AS theday, S.n AS shiftno
FROM dbo.Nums AS D
  CROSS JOIN dbo.Nums AS S
WHERE D.n <= 7
  AND S.N <= 3
ORDER BY theday, shiftno;
```

Here's the output of this query:

```
theday      shiftno
----------- -----------
1           1
1           2
1           3
2           1
2           2
2           3
3           1
3           2
3           3
4           1
4           2
4           3
5           1
5           2
5           3
6           1
6           2
6           3
7           1
7           2
7           3
```

The Nums table has 100,000 rows. According to logical query processing, the first step in the processing of the query is evaluating the FROM clause. The cross join operates in the FROM clause, performing a Cartesian product between the two instances of Nums, yielding a table with 10,000,000,000 rows (not to worry, that's only conceptually). Then the WHERE clause filters only the rows where the column D.n is less than or equal to 7, and the column S.n is less than or equal to 3. After applying the filter, the result has 21 qualifying rows. The SELECT clause then returns D.n aliasing it theday, and S.n aliasing it shiftno.

Fortunately, SQL Server doesn't have to follow logical query processing literally as long as it can return the correct result. That's what optimization is all about—returning the result as fast as possible. SQL Server knows that with a cross join followed by a filter it can evaluate the filters first (which is especially efficient when there are indexes to support the filters), and then match the remaining rows. This optimization technique is called *predicate pushdown*.

Note the importance of aliasing the tables in the join. For one, it's convenient to refer to a table by using a shorter name. But in a self-join like ours, table aliasing is mandatory. If you don't assign different aliases to the two instances of the table, you end up with an invalid result because there are duplicate column names even when including the table name as a prefix. By aliasing the tables differently, you can refer to columns in an unambiguous way using the form table_alias.column_name, as in D.n versus S.n.

Also, note that in addition to supporting the syntax for cross joins with the CROSS JOIN keywords, both standard SQL and T-SQL support an older syntax where you specify a comma between the table names, as in FROM T1, T2. However, for a number of reasons, it is recommended to stick to the newer syntax; it is less prone to errors and allows for more consistent code.

Inner joins

With an inner join, you can match rows from two tables based on a predicate—usually one that compares a primary key value in one side to a foreign key value in another side. Figure 1-7 illustrates an inner join.

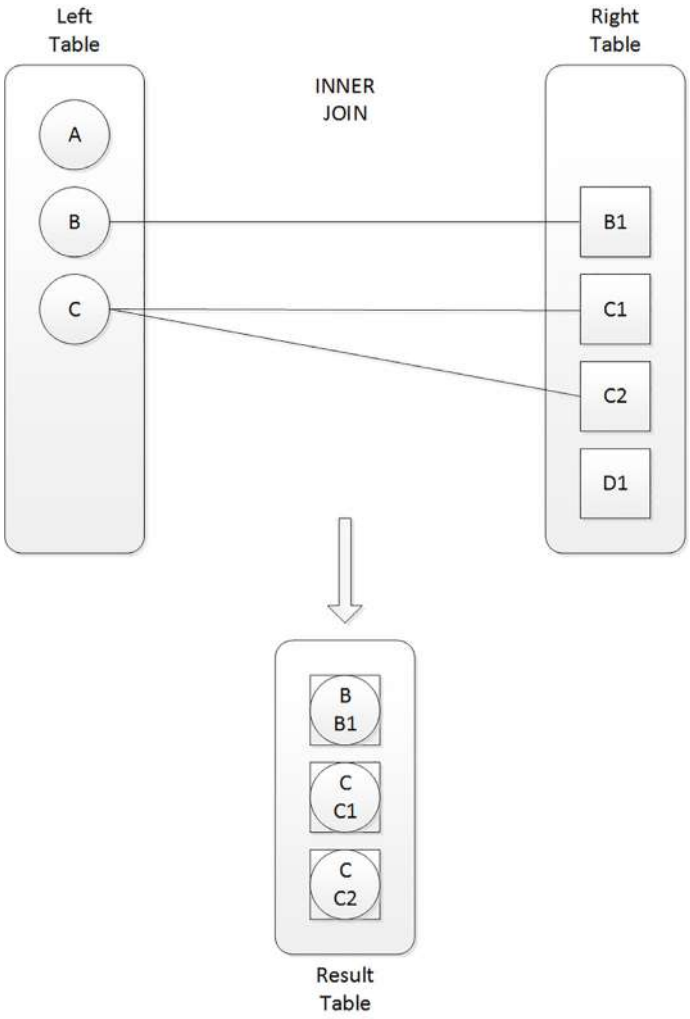


FIGURE 1-7 Inner join

The letters represent primary key values in the left table and foreign key values in the right table. Assuming the join is an equijoin (using a predicate with an equality operator such as lefttable.keycol = righttable.keycol), the inner join returns only matching rows for which the predicate evaluates to true. Rows for which the predicate evaluates to false or unknown are discarded.

As an example, the following query returns suppliers from Japan and the products they supply:

```
SELECT
  S.companyname AS supplier, S.country,
  P.productid, P.productname, P.unitprice
FROM Production.Suppliers AS S
```

```
  INNER JOIN Production.Products AS P
    ON S.supplierid = P.supplierid
WHERE S.country = N'Japan';
```

Here's the output of this query:

```
supplier        country  productid   productname    unitprice
--------------- -------- ----------- -------------- ----------
Supplier QOVFD  Japan    9           Product AOZBW  97.00
Supplier QOVFD  Japan    10          Product YHXGE  31.00
Supplier QWUSF  Japan    13          Product POXFU  6.00
Supplier QWUSF  Japan    14          Product PWCJB  23.25
Supplier QWUSF  Japan    15          Product KSZOI  15.50
Supplier QOVFD  Japan    74          Product BKAZJ  10.00
```

Observe that the join's matching predicate is specified in the ON clause. It matches suppliers and products that share the same supplier ID. Rows from either side that don't find a match in the other are discarded. For example, suppliers from Japan with no related products aren't returned.

> ***NOTE*** **INDEXING FOREIGN KEY COLUMNS**
>
> **Often, when joining tables, you join them based on a foreign key–unique key relationship. For example, there's a foreign key defined on the supplierid column in the Production.Products table (the referencing table), referencing the primary key column supplierid in the Production.Suppliers table (the referenced table). It's also important to note that when you define a primary key or unique constraint, SQL Server creates a unique index on the constraint columns to enforce the constraint's uniqueness property. But when you define a foreign key, SQL Server doesn't create any indexes on the foreign key columns. Such indexes could improve the performance of joins based on those relationships. Because SQL Server doesn't create such indexes automatically, it's your responsibility to identify the cases where they can be useful and create them. So when working on index tuning, one interesting area to examine is foreign key columns, and evaluating the benefits of creating indexes on those.**

Regarding the last query, again, notice the convenience of using short table aliases when needing to refer to ambiguous column names like supplierid. Observe that the query uses table aliases to prefix even nonambiguous column names such as s.country. This practice isn't mandatory as long as the column name is not ambiguous, but it is still considered a best practice for clarity.

A very common question is, "What's the difference between the ON and the WHERE clauses, and does it matter if you specify your predicate in one or the other?" The answer is that for inner joins it doesn't matter. Both clauses serve the same filtering purpose. Both filter only rows for which the predicate evaluates to true and discard rows for which the predicate evaluates to false or unknown. In terms of logical query processing, the WHERE is evaluated right after the FROM, so conceptually it is equivalent to concatenating the predicates with an AND operator, forming a conjunction of predicates. SQL Server knows this, and therefore can

internally rearrange the order in which it evaluates the predicates in practice, and it does so based on cost estimates.

For these reasons, if you wanted, you could rearrange the placement of the predicates from the previous query, specifying both in the ON clause, and still retain the original meaning, as follows:

```
SELECT
  S.companyname AS supplier, S.country,
  P.productid, P.productname, P.unitprice
FROM Production.Suppliers AS S
  INNER JOIN Production.Products AS P
    ON S.supplierid = P.supplierid
    AND S.country = N'Japan';
```

For many people, though, it's intuitive to specify the predicate that matches columns from both sides in the ON clause, and predicates that filter columns from only one side in the WHERE clause. Also think about the potential that in the future you will need to change an inner join to an outer join, where there's a difference between the roles that the ON and WHERE clauses play, as I describe in the next section. But again, with inner joins it doesn't matter. In the discussion of outer joins in the next section, you will see that, with those, ON and WHERE play different roles; you need to figure out, according to your needs, which is the appropriate clause for each of your predicates.

As another example for an inner join, the following query joins two instances of the HR.Employees table to match employees with their managers (a manager is also an employee, hence the self-join):

```
SELECT E.empid,
  E.firstname + N' ' + E.lastname AS emp,
  M.firstname + N' ' + M.lastname AS mgr
FROM HR.Employees AS E
  INNER JOIN HR.Employees AS M
    ON E.mgrid = M.empid;
```

This query generates the following output:

```
empid       emp                           mgr
----------- ------------------------------ ------------------------------
2           Don Funk                      Sara Davis
3           Judy Lew                      Don Funk
4           Yael Peled                    Judy Lew
5           Sven Mortensen                Don Funk
6           Paul Suurs                    Sven Mortensen
7           Russell King                  Sven Mortensen
8           Maria Cameron                 Judy Lew
9           Patricia Doyle                Sven Mortensen
```

Observe the join predicate: ON E.mgrid = M.empid. The employee instance is aliased as E and the manager instance as M. To find the right matches, the employee's manager ID needs to be equal to the manager's employee ID.

Note that only eight rows were returned even though there are nine rows in the table. The reason is that the CEO (Sara Davis, employee ID 1) has no manager, and therefore, her mgrid column is NULL. Remember that an inner join does not return rows that don't find matches.

As with cross joins, both standard SQL and T-SQL support an older syntax for inner joins where you specify a comma between the table names, and then all predicates in the WHERE clause. But as mentioned, it is considered best practice to stick to the newer syntax with the JOIN keyword. When using the older syntax, if you forget to indicate the join predicate, you end up with an unintentional cross join. When using the newer syntax, an inner join isn't valid syntactically without an ON clause, so if you forget to indicate the join predicate, the parser will generate an error.

Because an inner join is the most commonly used type of join, the standard decided to make it the default in case you specify just the JOIN keyword. So T1 JOIN T2 is equivalent to T1 INNER JOIN T2.

Outer joins

With outer joins, you can request to preserve all rows from one or both sides of the join, never mind if there are matching rows in the other side based on the ON predicate.

By using the keywords LEFT OUTER JOIN (or LEFT JOIN for short), you ask to preserve the left table. The join returns what an inner join normally would—that is, matches (call those inner rows). In addition, the join also returns rows from the left table that have no matches in the right table (call those outer rows), with NULLs used as placeholders in the right side. Figure 1-8 shows an illustration of a left outer join.

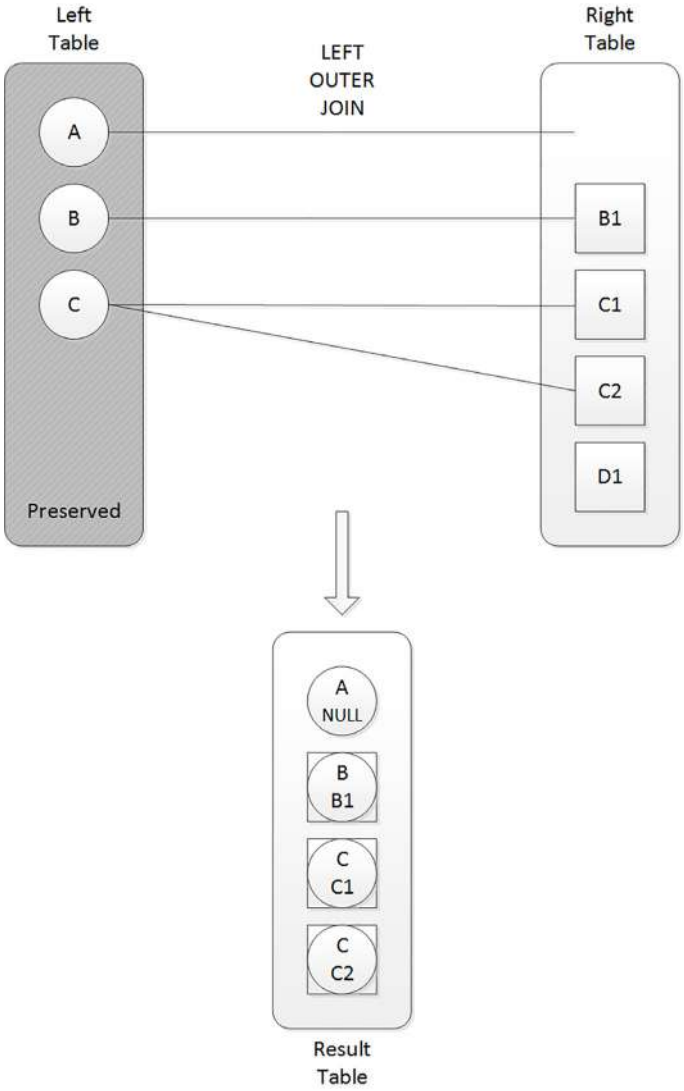


FIGURE 1-8 Left outer join

Unlike in the inner join, the left row with the key A is returned even though it has no match in the right side. It's returned with NULLs as placeholders in the right side.

As an example, the following query returns suppliers from Japan and the products they supply, including suppliers from Japan that don't have related products.

```
SELECT
  S.companyname AS supplier, S.country,
  P.productid, P.productname, P.unitprice
FROM Production.Suppliers AS S
  LEFT OUTER JOIN Production.Products AS P
    ON S.supplierid = P.supplierid
WHERE S.country = N'Japan';
```

This query generates the following output:

```
supplier        country  productid   productname    unitprice
--------------- -------- ----------- -------------- ----------
Supplier QOVFD  Japan    9           Product AOZBW  97.00
Supplier QOVFD  Japan    10          Product YHXGE  31.00
Supplier QOVFD  Japan    74          Product BKAZJ  10.00
Supplier QWUSF  Japan    13          Product POXFU  6.00
Supplier QWUSF  Japan    14          Product PWCJB  23.25
Supplier QWUSF  Japan    15          Product KSZOI  15.50
Supplier XYZ    Japan    NULL        NULL           NULL
```

Because the Production.Suppliers table is the preserved side of the join, Supplier XYZ is returned even though it has no matching products. As you recall, an inner join did not return this supplier.

It is very important to understand that, with outer joins, the ON and WHERE clauses play very different roles, and therefore, they aren't interchangeable. The WHERE clause still plays a simple filtering role—namely, it keeps true cases and discards false and unknown cases. In our query, the WHERE clause filters only suppliers from Japan, so suppliers that aren't from Japan simply don't show up in the output.

However, the ON clause doesn't play a simple filtering role; rather, it's a more sophisticated matching role. In other words, a row in the preserved side will be returned whether the ON predicate finds a match for it or not. So the ON predicate only determines which rows from the nonpreserved side get matched to rows from the preserved side—not whether to return the rows from the preserved side. In our query, the ON clause matches rows from both sides by comparing their supplier ID values. Because it's a matching predicate (as opposed to a filter), the join won't discard suppliers; instead, it only determines which products get matched to each supplier. But even if a supplier has no matches based on the ON predicate, the supplier is still returned. In other words, ON is not final with respect to the preserved side of the join. WHERE is final. So when in doubt, whether to specify the predicate in the ON or WHERE clauses, ask yourself: Is the predicate used to filter or match? Is it supposed to be final or nonfinal?

With this in mind, guess what happens if you specify both the predicate that compares the supplier IDs from both sides, and the one comparing the supplier country to Japan in the ON clause? Try it.

```
SELECT
  S.companyname AS supplier, S.country,
  P.productid, P.productname, P.unitprice
FROM Production.Suppliers AS S
  LEFT OUTER JOIN Production.Products AS P
    ON S.supplierid = P.supplierid
   AND S.country = N'Japan';
```

Observe what's different in the result (shown here in abbreviated form) and see if you can explain in your own words what the query returns now:

```
supplier        country  productid  productname    unitprice
--------------- -------- ---------- -------------- ----------
Supplier SWRXU  UK       NULL       NULL           NULL
Supplier VHQZD  USA      NULL       NULL           NULL
Supplier STUAZ  USA      NULL       NULL           NULL
Supplier QOVFD  Japan    9          Product AOZBW  97.00
Supplier QOVFD  Japan    10         Product YHXGE  31.00
Supplier QOVFD  Japan    74         Product BKAZJ  10.00
Supplier EQPNC  Spain    NULL       NULL           NULL
Supplier QWUSF  Japan    13         Product POXFU  6.00
Supplier QWUSF  Japan    14         Product PWCJB  23.25
Supplier QWUSF  Japan    15         Product KSZOI  15.50
...

(34 row(s) affected)
```

Now that both predicates appear in the ON clause, both serve a matching purpose. What this means is that all suppliers are returned—even those that aren't from Japan. But in order to match a product to a supplier, the supplier IDs in both sides need to match, and the supplier country needs to be Japan.

Just like you can use a left outer join to preserve the left side, you can use a right outer join to preserve the right side. Use the keywords RIGHT OUTER JOIN (or RIGHT JOIN in short). Figure 1-9 shows an illustration of a right outer join.

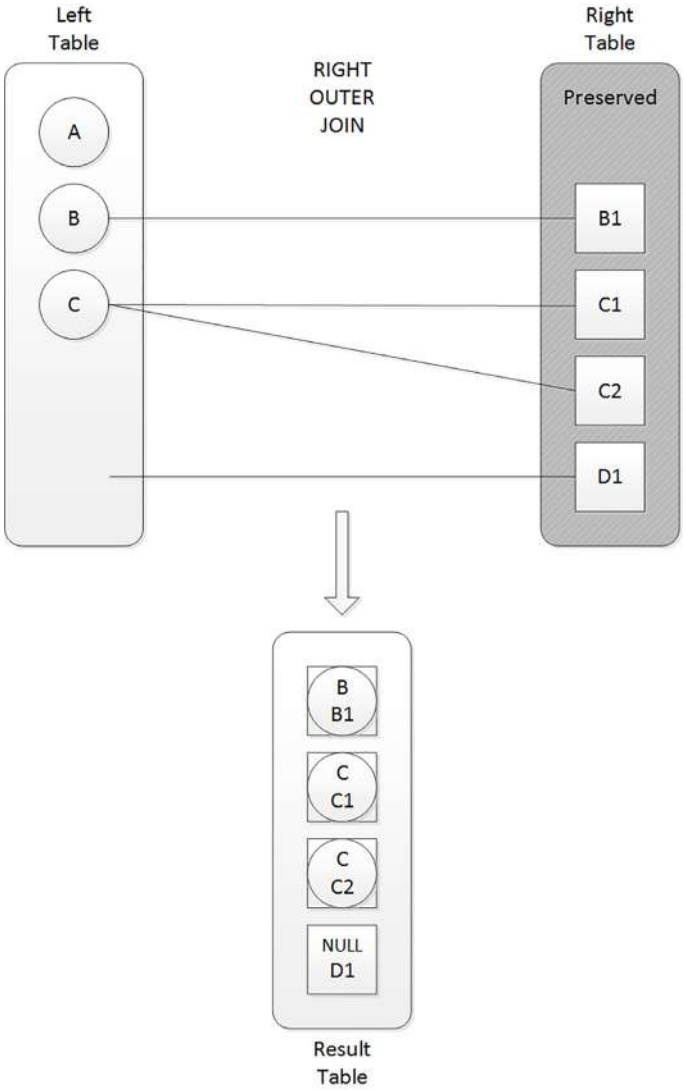


FIGURE 1-9 Right outer join

T-SQL also supports a full outer join (FULL OUTER JOIN, or FULL JOIN in short) that preserves both sides. Figure 1-10 shows an illustration of this type of join.

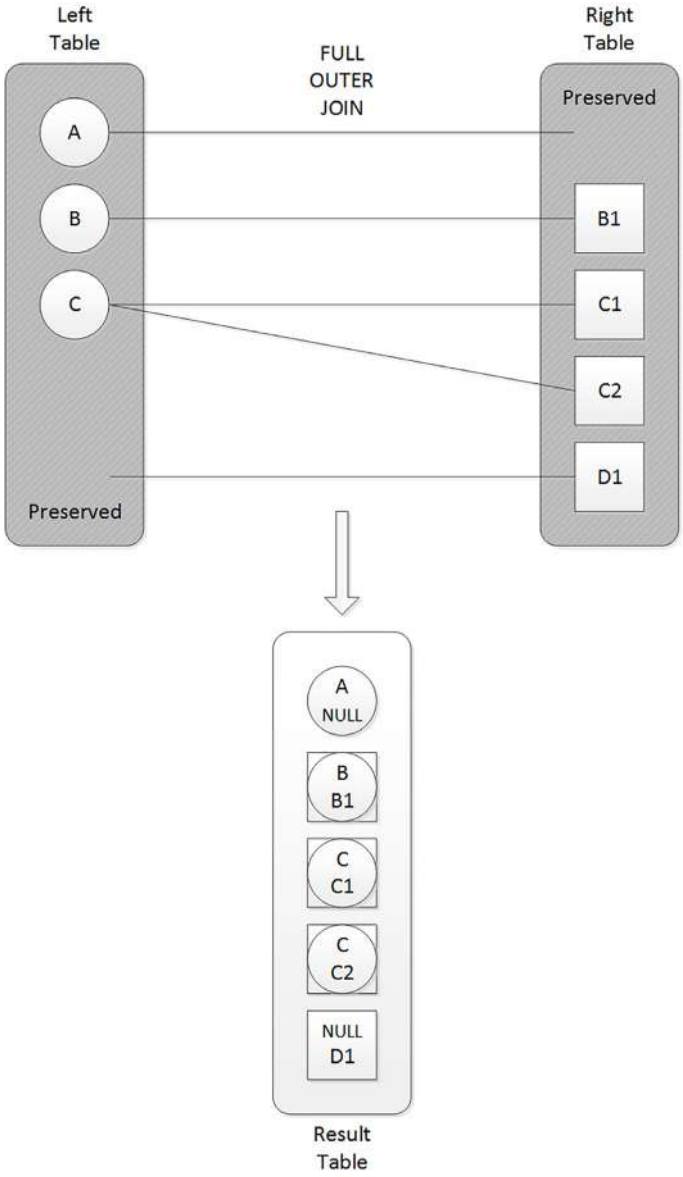


FIGURE 1-10 Full outer join

A full outer join returns the matched rows, which are normally returned from an inner join; plus rows from the left that don't have matches in the right, with NULLs used as placeholders in the right side; plus rows from the right that don't have matches in the left, with NULLs used as placeholders in the left side. It's not common to need a full outer join because most relationships between tables allow only one of the sides to have rows that don't have matches in the other, in which case, a one-sided outer join is needed.

Queries with composite joins and NULLs in join columns

Some joins can be a bit tricky to handle, for instance when the join columns can have NULLs, or when you have multiple join columns—what's known as a composite join. This section focuses on such cases.

Earlier in the inner joins section is a query that matched employees and their managers. Remember that the inner join eliminated the CEO's row because the mgrid is NULL in that row, and therefore the join found no matching manager. If you want to include the CEO's row, you need to use an outer join to preserve the side representing the employees (E) as follows:

```
SELECT E.empid,
  E.firstname + N' ' + E.lastname AS emp,
  M.firstname + N' ' + M.lastname AS mgr
FROM HR.Employees AS E
  LEFT OUTER JOIN HR.Employees AS M
    ON E.mgrid = M.empid;
```

Here's the output of this query, this time including the CEO's row:

```
empid       emp                           mgr
----------- ------------------------------ ------------------------------
1           Sara Davis                    NULL
2           Don Funk                      Sara Davis
3           Judy Lew                      Don Funk
4           Yael Peled                    Judy Lew
5           Sven Mortensen                Don Funk
6           Paul Suurs                    Sven Mortensen
7           Russell King                  Sven Mortensen
8           Maria Cameron                 Judy Lew
9           Patricia Doyle                Sven Mortensen
```

As a reminder, the order of the output is not guaranteed unless you add an ORDER BY clause to the query. This means that theoretically you can have the results returned in a different order than mine.

When you need to join tables that are related based on multiple columns, the join is called a *composite join* and the ON clause typically consists of a conjunction of predicates (predicates separated by AND operators) that match the corresponding columns from the two sides. Sometimes you need more complex predicates, especially when NULLs are involved. I'll demonstrate this by using a pair of tables. One table is called EmpLocations and it holds employee locations and the number of employees in each location. Another table is called CustLocations and it holds customer locations and the number of customers in each location. Run the following code to create these tables and populate them with sample data:

```
DROP TABLE IF EXISTS dbo.EmpLocations;

SELECT country, region, city, COUNT(*) AS numemps
INTO dbo.EmpLocations
FROM HR.Employees
GROUP BY country, region, city;

ALTER TABLE dbo.EmpLocations ADD CONSTRAINT UNQ_EmpLocations
  UNIQUE CLUSTERED(country, region, city);
```

```
DROP TABLE IF EXISTS dbo.CustLocations;

SELECT country, region, city, COUNT(*) AS numcusts
INTO dbo.CustLocations
FROM Sales.Customers
GROUP BY country, region, city;

ALTER TABLE dbo.CustLocations ADD CONSTRAINT UNQ_CustLocations
  UNIQUE CLUSTERED(country, region, city);
```

There's a key defined in both tables on the location attributes: country, region, and city. Instead of using a primary key constraint I used a unique constraint to enforce the key because the region attribute allows NULLs, and between the two types of constraints, only the latter allows NULLs. I also specified the CLUSTERED keyword in the unique constraint definitions to have SQL Server create a clustered index type to enforce the constraint's uniqueness property. This index will be beneficial in supporting joins between the tables based on the location attributes as well filters based on those attributes.

Query the EmpLocations table to see its contents:

```
SELECT country, region, city, numemps
FROM dbo.EmpLocations;
```

This query generates the following output:

```
country         region          city            numemps
--------------- --------------- --------------- -----------
UK              NULL            London          4
USA             WA              Kirkland        1
USA             WA              Redmond         1
USA             WA              Seattle         2
USA             WA              Tacoma          1
```

Query the CustLocations table:

```
SELECT country, region, city, numcusts
FROM dbo.CustLocations;
```

This query generates the following output, shown here in abbreviated form:

```
country         region          city            numcusts
--------------- --------------- --------------- -----------
Argentina       NULL            Buenos Aires    3
Austria         NULL            Graz            1
Austria         NULL            Salzburg        1
Belgium         NULL            Bruxelles       1
Belgium         NULL            Charleroi       1
Brazil          RJ              Rio de Janeiro  3
Brazil          SP              Campinas        1
Brazil          SP              Resende         1
Brazil          SP              Sao Paulo       4
Canada          BC              Tsawassen       1
...

(69 row(s) affected)
```

Suppose that you needed to join the two tables returning only matched locations, with both the employee and customer counts returned along with the location attributes. Your first attempt might be to write a composite join with an ON clause that has a conjunction of simple equality predicates as follows:

```
SELECT EL.country, EL.region, EL.city, EL.numemps, CL.numcusts
FROM dbo.EmpLocations AS EL
  INNER JOIN dbo.CustLocations AS CL
    ON EL.country = CL.country
    AND EL.region = CL.region
    AND EL.city = CL.city;
```

This query generates the following output:

```
country         region          city            numemps     numcusts
--------------- --------------- --------------- ----------- -----------
USA             WA              Kirkland        1           1
USA             WA              Seattle         2           1
```

The problem is that the region column supports NULLs representing cases where the region is irrelevant (missing but inapplicable) and when you compare NULLs with an equality-based predicate the result is the logical value unknown, in which case the row is discarded. For instance, the location UK, NULL, London appears in both tables, and therefore you expect to see it in the result of the join, but you don't. A common way for people to resolve this problem is to use the ISNULL or COALESCE functions to substitute a NULL in both sides with a value that can't normally appear in the data, and this way when both sides are NULL you get a true back from the comparison. Here's an example for implementing this solution using the ISNULL function:

```
SELECT EL.country, EL.region, EL.city, EL.numemps, CL.numcusts
FROM dbo.EmpLocations AS EL
  INNER JOIN dbo.CustLocations AS CL
    ON EL.country = CL.country
    AND ISNULL(EL.region, N'<N/A>') = ISNULL(CL.region, N'<N/A>')
    AND EL.city = CL.city;
```

This time the query generates the correct result:

```
country         region          city            numemps     numcusts
--------------- --------------- --------------- ----------- -----------
UK              NULL            London          4           6
USA             WA              Kirkland        1           1
USA             WA              Seattle         2           1
```

The problem with this approach is that once you apply manipulation to a column, SQL Server cannot trust that the result values preserve the same ordering behavior as the original values. This can negatively affect the ability of SQL Server to rely on index ordering when it optimizes the query. Our query gets the plan shown in Figure 1-11.

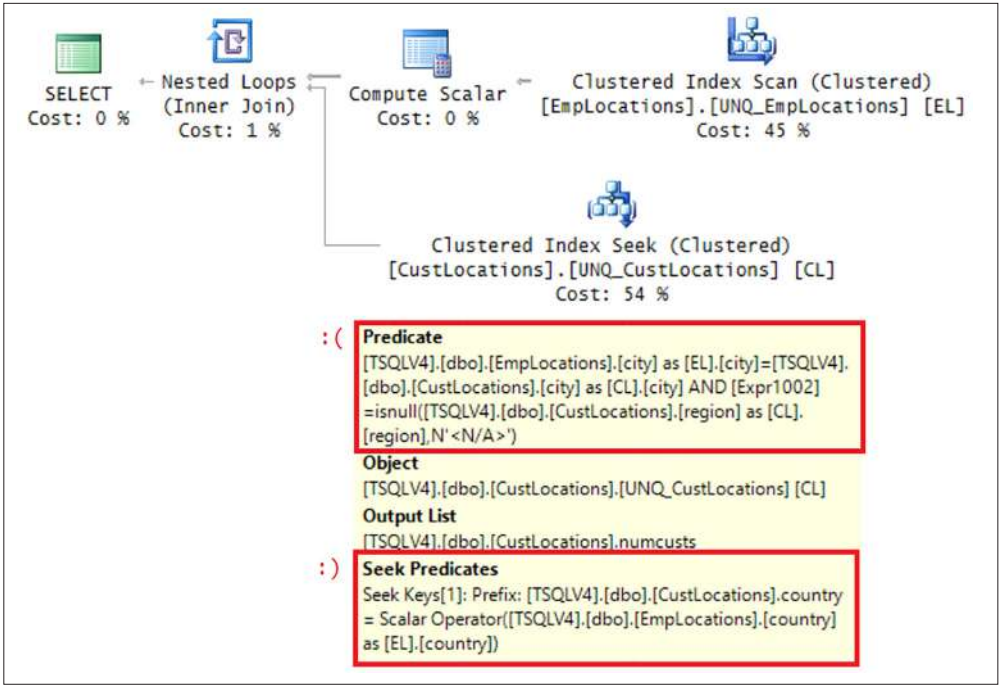


FIGURE 1-11 Plan for query with ISNULL

The plan scans the clustered index on EmpLocations, and for each row (employee location) performs a seek in the clustered index on CustLocations. However, notice that the seek relies on only the country attribute in the seek predicate. It cannot rely on the region and city attributes because of the manipulation that you applied to the region attribute. The predicates involving the region and city attributes appear as residual predicates (under the Predicate property). This means that for each employee location row, the Clustered Index Seek operator that is applied to the CustLocations index performs a range scan of the entire customer location's country that is equal to the current employee location's country. The residual predicates that are based on region and city then determine whether to keep or discard each row. That's a lot of unnecessary effort.

The optimizer picked the nested loops strategy in the plan shown in Figure 1-11 because the sample tables that we used are so tiny. With bigger, more realistic, table sizes, the optimizer typically chooses a merge join algorithm when the data is preordered by the join columns in both sides. This algorithm processes both sides of the join based on join column order, and in a way, zips matching rows together. The data can either be pulled preordered from an index, or explicitly sorted. As mentioned, applying manipulation to join columns breaks the ordering property of the data, and therefore even if it's preordered in an index, the optimizer cannot trust this order. To illustrate how this can affect the merge algorithm, force it in our query by adding the MERGE join hint as follows.

```
SELECT EL.country, EL.region, EL.city, EL.numemps, CL.numcusts
FROM dbo.EmpLocations AS EL
  INNER MERGE JOIN dbo.CustLocations AS CL
    ON EL.country = CL.country
    AND ISNULL(EL.region, N'<N/A>') = ISNULL(CL.region, N'<N/A>')
    AND EL.city = CL.city;
```

The plan for this query is shown in Figure 1-12.

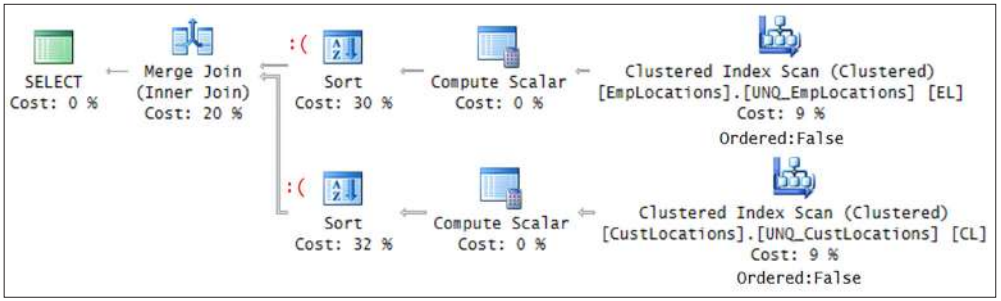


FIGURE 1-12 Plan for query with ISNULL and MERGE algorithm

Observe that the clustered indexes on both tables are scanned in an Orderd: False fashion, meaning that the scan is not requested to return the data in index order. Then the join columns sort both sides explicitly before being merged.

You can handle NULLs in a manner that gives you the desired logical meaning and that at the same time is considered order preserving by the optimizer using the predicate: (EL.region = CL.region OR (EL.region IS NULL AND CL.region IS NULL)). Here's the complete solution query:

```
SELECT EL.country, EL.region, EL.city, EL.numemps, CL.numcusts
FROM dbo.EmpLocations AS EL
  INNER JOIN dbo.CustLocations AS CL
    ON EL.country = CL.country
    AND (EL.region = CL.region OR (EL.region IS NULL AND CL.region IS NULL))
    AND EL.city = CL.city;
```

The plan for this query is shown in Figure 1-13.

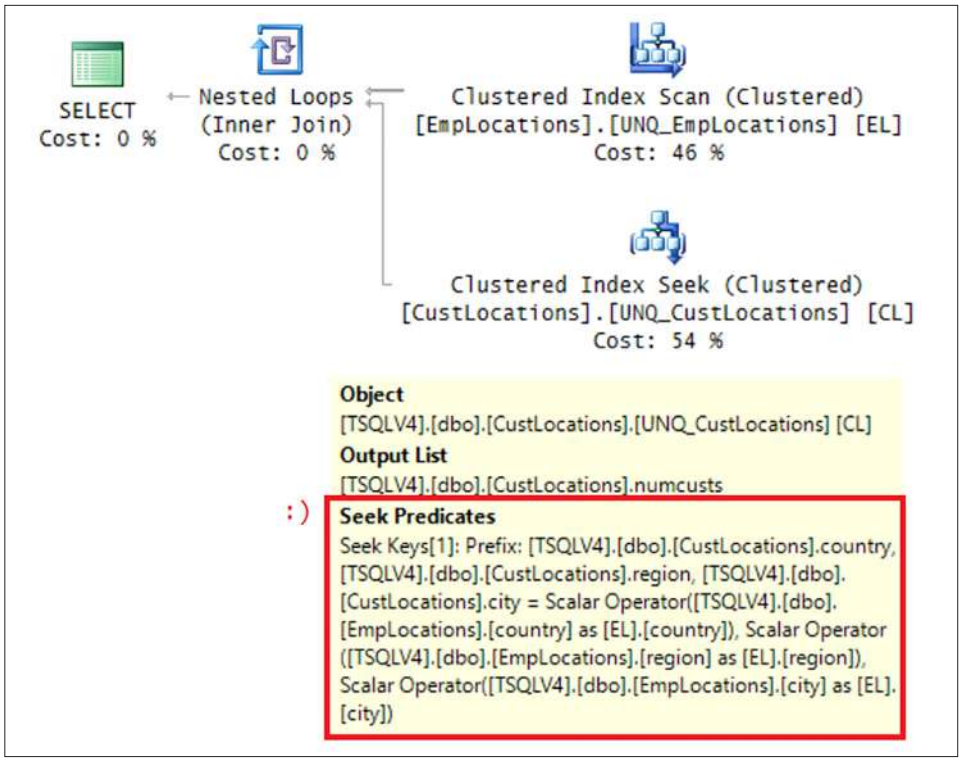


FIGURE 1-13 Plan with order preservation

Notice that this time all predicates show up as seek predicates.

Similarly, with the new predicate, the optimizer can rely on index order when using the merge join algorithm. To demonstrate this, again, force this algorithm by adding the MERGE join hint as follows:

```
SELECT EL.country, EL.region, EL.city, EL.numemps, CL.numcusts
FROM dbo.EmpLocations AS EL
  INNER MERGE JOIN dbo.CustLocations AS CL
    ON EL.country = CL.country
    AND (EL.region = CL.region OR (EL.region IS NULL AND CL.region IS NULL))
    AND EL.city = CL.city;
```

The plan for this query is shown in Figure 1-14.

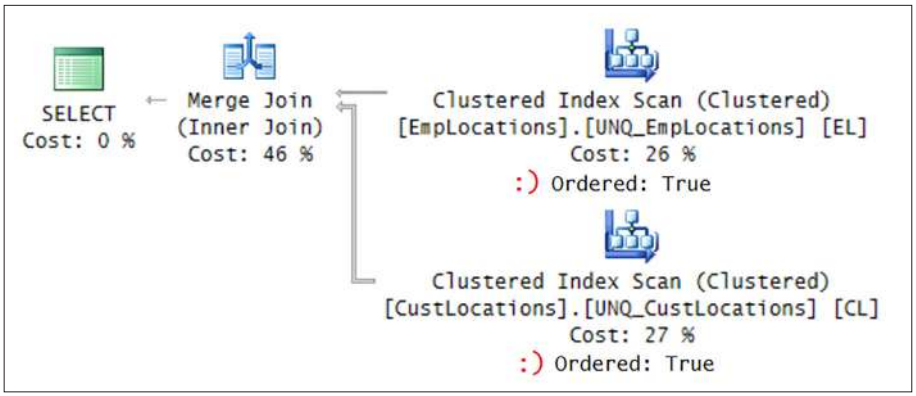


FIGURE 1-14 Plan with order preservation and MERGE algorithm

Observe that the plan scans both clustered indexes in order, and that there's no explicit sorting taking place prior to the merge join.

Recall that when set operators combine query results they compare corresponding attributes using distinctness and not equality, producing true when comparing two NULLs. However, one drawback that set operators have is that they compare complete rows. Unlike joins, which allow comparing a subset of the attributes and return additional ones in the result, set operators must compare all attributes from the two input queries. But in T-SQL, you can combine joins and set operators to benefit from the advantages of both tools. Namely, rely on the distinctness-based comparison of set operators and the ability of joins to return additional attributes beyond what you compare. In our querying task, the solution looks like this:

```
SELECT EL.country, EL.region, EL.city, EL.numemps, CL.numcusts
FROM dbo.EmpLocations AS EL
  INNER JOIN dbo.CustLocations AS CL
    ON EXISTS (SELECT EL.country, EL.region, EL.city
               INTERSECT
               SELECT CL.country, CL.region, CL.city);
```

For each row that is evaluated by the join, the set operator performs an intersection of the employee location attributes and customer location attributes using FROM-less SELECT statements, each producing one row. If the locations intersect, the result is one row, in which case the EXISTS predicate returns true, and the evaluated row is considered a match. If the locations don't intersect, the result is an empty set, in which case the EXISTS predicate returns false, and the evaluated row is not considered a match. Remarkably, Microsoft added logic to the optimizer to consider this form order-preserving. The plan for this query is the same as the one shown earlier in Figure 1-13.

Use the following code to force the merge algorithm in the query:

```
SELECT EL.country, EL.region, EL.city, EL.numemps, CL.numcusts
FROM dbo.EmpLocations AS EL
  INNER MERGE JOIN dbo.CustLocations AS CL
    ON EXISTS (SELECT EL.country, EL.region, EL.city
```

```
INTERSECT
SELECT CL.country, CL.region, CL.city);
```

Also here the ordering property of the data is preserved and you get the plan shown earlier in Figure 1-14, where the clustered indexes of both sides are scanned in order and there's no need for explicit sorting prior to performing the merge join.

When you're done, run the following code for cleanup:

```
DROP TABLE IF EXISTS dbo.CustLocations;
DROP TABLE IF EXISTS dbo.EmpLocations;
```

Multi-join queries

It's important to remember that a join in T-SQL takes place conceptually between two tables at a time. A multi-join query evaluates the joins conceptually from left to right. So the result of one join is used as the left input to the next join. If you don't understand this, you can end up with logical bugs, especially when outer joins are involved. (With inner and cross-joins, the order cannot affect the meaning.)

As an example, suppose that you wanted to return all suppliers from Japan, and matching products where relevant. For this, you need an outer join between Production.Suppliers and Production.Products, preserving Suppliers. But you also want to include product category information, so you add an inner join to Production.Categories, as follows:

```
SELECT
  S.companyname AS supplier, S.country,
  P.productid, P.productname, P.unitprice,
  C.categoryname
FROM Production.Suppliers AS S
  LEFT OUTER JOIN Production.Products AS P
    ON S.supplierid = P.supplierid
  INNER JOIN Production.Categories AS C
    ON C.categoryid = P.categoryid
WHERE S.country = N'Japan';
```

This query generates the following output:

```
supplier        country productid  productname    unitprice  categoryname
--------------- ------- ---------- -------------- ---------- -------------
Supplier QOVFD  Japan   9          Product AOZBW  97.00      Meat/Poultry
Supplier QOVFD  Japan   10         Product YHXGE  31.00      Seafood
Supplier QWUSF  Japan   13         Product POXFU  6.00       Seafood
Supplier QWUSF  Japan   14         Product PWCJB  23.25      Produce
Supplier QWUSF  Japan   15         Product KSZOI  15.50      Condiments
Supplier QOVFD  Japan   74         Product BKAZJ  10.00      Produce
```

Supplier XYZ from Japan was discarded. Can you explain why?

Conceptually, the first join included outer rows (suppliers with no products) but produced NULLs as placeholders in the product attributes in those rows. Then the join to Production.Categories compared the NULLs in the categoryid column in the outer rows to categoryid

values in Production.Categories, and discarded those rows. In short, the inner join that followed the outer join nullified the outer part of the join. In fact, if you look at the query plan for this query, you will find that the optimizer didn't even bother to process the join between Production.Suppliers and Production.Products as an outer join. It detected the contradiction between the outer join and the subsequent inner join, and converted the first join to an inner join too.

There are a number of ways to address this problem. One is to use a LEFT OUTER in both joins, like so:

```
SELECT
  S.companyname AS supplier, S.country,
  P.productid, P.productname, P.unitprice,
  C.categoryname
FROM Production.Suppliers AS S
  LEFT OUTER JOIN Production.Products AS P
    ON S.supplierid = P.supplierid
  LEFT OUTER JOIN Production.Categories AS C
    ON C.categoryid = P.categoryid
WHERE S.country = N'Japan';
```

Another option is to use an interesting capability in the language—separate some of the joins to their own independent logical phase. What you're after is a left outer join between Production.Suppliers and the result of the inner join between Production.Products and Production.Categories. You can phrase your query exactly like this:

```
SELECT
  S.companyname AS supplier, S.country,
  P.productid, P.productname, P.unitprice,
  C.categoryname
FROM Production.Suppliers AS S
  LEFT OUTER JOIN
    (Production.Products AS P
       INNER JOIN Production.Categories AS C
         ON C.categoryid = P.categoryid)
    ON S.supplierid = P.supplierid
WHERE S.country = N'Japan';
```

With both fixes, the query generates the correct result, including suppliers who currently don't have related products:

```
supplier        country  productid  productname    unitprice  categoryname
--------------- -------- ---------- -------------- ---------- -------------
Supplier QOVFD  Japan    9          Product AOZBW  97.00      Meat/Poultry
Supplier QOVFD  Japan    10         Product YHXGE  31.00      Seafood
Supplier QOVFD  Japan    74         Product BKAZJ  10.00      Produce
Supplier QWUSF  Japan    13         Product POXFU  6.00       Seafood
Supplier QWUSF  Japan    14         Product PWCJB  23.25      Produce
Supplier QWUSF  Japan    15         Product KSZOI  15.50      Condiments
Supplier XYZ    Japan    NULL       NULL           NULL       NULL
```

Note that the important change that made the difference is the arrangement of the ON clauses with respect to the joined tables. The ON clause ordering is what defines the logi-

cal join ordering. Each ON clause must appear right below the two units that it joins. By specifying the ON clause that matches attributes from Production.Products and Production.Categories first, you set this inner join to be logically evaluated first. Then the second ON clause handles the left outer join by matching an attribute from Production.Suppliers with an attribute from the result of the inner join. Curiously, T-SQL doesn't really require the parentheses that I added to the query; remove those and rerun the query, and you will see that it runs successfully. However, it's recommended to use those for clarity.

EXAM TIP

Multi join queries that mix different join types are very common in practice and therefore there's a high likelihood that questions about those will show up in the exam. Make sure that you understand the pitfalls in mixing join types, especially when an outer join is subsequently followed by an inner join, which discards the outer rows that were produced by the outer join.

When you're done, run the following code to delete the supplier row that you added at the beginning of this skill:

```
DELETE FROM Production.Suppliers WHERE supplierid > 29;
```

Skill 1.3: Implement functions and aggregate data

T-SQL supports many built-in functions that you can use to manipulate data. Scalar-valued functions return a single value and table-valued functions return a table result. Use of built-in functions can improve developer productivity, but you also need to understand cases where their use in certain context can end up negatively affecting query performance. It's also important to understand the concept of function determinism and its effects on your queries.

Note that this skill is not meant to be an exhaustive coverage of all functions that T-SQL supports—this would require a whole book in its own right. Instead, this chapter explains key aspects of working with functions, usually in the context of certain types of data, like date and time data, or character data. For more details about built-in functions, see the topic "Built-in Functions (Transact-SQL)" at *http://msdn.microsoft.com/en-us/library/ms174318(v=SQL.110).aspx*.

This section covers how to:

- Construct queries using scalar-valued and table-valued functions
- Identify the impact of function usage to query performance and WHERE clause sargability
- Identify the differences between deterministic and non-deterministic functions
- Use built-in aggregate functions
- Use arithmetic functions, date-related functions, and system functions

Type conversion functions

T-SQL supports a number of functions that can convert a source expression to a target data type. In my examples I use constants as the source values to demonstrate the use of the functions, but typically you apply such functions to columns or expressions based on columns as part of a query.

The two fundamental functions that T-SQL supports for conversion purposes are CAST and CONVERT. The former is standard whereas the latter is proprietary in T-SQL. The CAST function's syntax is CAST(source_expression AS target_type. For example, CAST('100' AS INT) converts the source character string constant to the target integer value 100. The CONVERT function is handy when you need to specify a style for the conversion. Its syntax is CONVERT(target_type, source_expression [, style_number]). You can find the supported style numbers and their meaning at *https://msdn.microsoft.com/en-us/library/ms187928.aspx*. For instance, when converting a character string to a date and time type or the other way around, you can specify the style number to avoid ambiguity in case the form you use is considered language dependent. As an example, the expression CONVERT(DATE, '01/02/2017', 101) converts the input string to a date using the U.S. style, meaning January 2, 2017. The expression CONVERT(DATE, '01/02/2017', 103) uses the British/French style, meaning February 1, 2017.

The PARSE function is an alternative to CONVERT when you want to parse a character string input to a target type, but instead of using cryptic style numbers, it uses a more user-friendly .NET culture name. For instance, the expression PARSE('01/02/2017' AS DATE USING 'en-US') uses the English US culture, parsing the input as a date meaning January 2, 2017. The expression PARSE('01/02/2017' AS DATE USING 'en-GB') uses the English Great Britain culture, parsing the input as a date meaning February 1, 2017. Note though that this function is significantly slower than CONVERT, so I personally stay away from using it.

One of the problems with CAST, CONVERT, and PARSE is that if the function fails to convert a value within a query, the whole query fails and stops processing. As an alternative to these functions, T-SQL supports the TRY_CAST, TRY_CONVERT, and TRY_PARSE functions, which behave the same as their counterparts when the conversion is valid, but return a NULL instead of failing when the conversion isn't valid. As an example, run the following code to try and convert two strings to dates using the CONVERT function:

```
SELECT CONVERT(DATE, '14/02/2017', 101) AS col1,
  CONVERT(DATE, '02/14/2017', 101) AS col2;
```

The first value doesn't convert successfully and therefore this code fails with the following error:

```
Msg 241, Level 16, State 1, Line 26
Conversion failed when converting date and/or time from character string.
```

Use the TRY_CONVERT function instead of CONVERT, like so:

```
SELECT TRY_CONVERT(DATE, '14/02/2017', 101) AS col1,
  TRY_CONVERT(DATE, '02/14/2017', 101) AS col2;
```

This time the code doesn't fail, but the first expression returns a NULL, as the following output shows:

```
col1       col2
---------- ----------
NULL       2017-02-14
```

Lastly, the FORMAT function is an alternative to the CONVERT function when you want to format an input expression of some type as a character string, but instead of using cryptic style numbers, you specify a .NET format string and culture, if relevant. For instance, to format an input date and time value, such as now, as a character string using the form 'yyyy-MM-dd', use the expression: FORMAT(SYSDATETIME(), 'yyyy-MM-dd'). You can use any format string supported by the .NET Framework. (For details, see the topics "FORMAT (Transact-SQL)" and "Formatting Types in the .NET Framework" at *https://msdn.microsoft.com/en-us/library/hh213505.aspx* and *http://msdn.microsoft.com/en-us/library/26etazsy.aspx*.). Note that like PARSE, the FORMAT function is also quite slow, so when you need to format a large number of values in a query, you typically get much better performance with alternative built-in functions.

Date and time functions

T-SQL supports a number of date and time functions that allow you to manipulate your date and time data. This section covers some of the important functions supported by T-SQL and provides some examples. For the full list, as well as the technical details and syntax, see the T-SQL documentation for the topic at *https://msdn.microsoft.com/en-us/library/ms186724.aspx*.

Current date and time

One important category of functions is the category that returns the current date and time.

The functions in this category are GETDATE, CURRENT_TIMESTAMP, GETUTCDATE, SYSDATETIME, SYSUTCDATETIME, and SYSDATETIMEOFFSET. GETDATE is T-SQL–specific, returning the current date and time in the SQL Server instance you're connected to as a DATETIME data type. CURRENT_TIMESTAMP is the same, only it's standard, and hence the recommended one to use. SYSDATETIME and SYSDATETIMEOFFSET are similar, only returning the values as the more precise DATETIME2 and DATETIMEOFFSET types (including the time zone offset from UTC), respectively. Note that there are no built-in functions to return the current date and the current time. To get such information, simply cast the SYSDATETIME function to DATE or TIME, respectively. For example, to get the current date, use CAST(SYSDATETIME() AS DATE). The GETUTCDATE function returns the current date and time in UTC terms as a DATETIME type, and SYSUTCDATETIME does the same, only returning the result as the more precise DATETIME2 type.

Date and time parts

This section covers date and time functions that either extract a part from a date and time value (like DATEPART) or construct a date and time value from parts (like DATEFROMPARTS).

Using the DATEPART function, you can extract from an input date and time value a desired part, such as a year, minute, or nanosecond, and return the extracted part as an integer. For example, the expression DATEPART(month, '20170212') returns 2. T-SQL provides the functions YEAR, MONTH, and DAY as abbreviations to DATEPART, not requiring you to specify the part. The DATENAME function is similar to DATEPART, only it returns the name of the part as a character string, as opposed to the integer value. Note that the function is language dependent. That is, if the effective language in your session is us_english, the expression DATENAME(month, '20170212') returns 'February', but for Italian, it returns 'febbraio.'

T-SQL provides a set of functions that construct a desired date and time value from its numeric parts. You have such a function for each of the six available date and time types: DATEFROMPARTS, DATETIME2FROMPARTS, DATETIMEFROMPARTS, DATETIMEOFFSETFROMPARTS, SMALLDATETIMEFROMPARTS, and TIMEFROMPARTS. For example, to build a DATE value from its parts, you would use an expression such as DATEFROMPARTS(2017, 02, 12).

Finally, the EOMONTH function computes the respective end of month date for the input date and time value. For example, suppose that today was February 12, 2017. The expression EOMONTH(SYSDATETIME()) would then return the date '2017-02-29'. This function supports a second optional input indicating how many months to add to the result (or subtract if negative).

Add and diff functions

T-SQL supports addition and difference date and time functions called DATEADD and DATEDIFF.

DATEADD is a very commonly used function. With it, you can add a requested number of units of a specified part to a specified date and time value. For example, the expression DATEADD(year, 1, '20170212') adds one year to the input date February 12, 2017.

DATEDIFF is another commonly used function; it returns the difference in terms of a requested part between two date and time values. For example, the expression DATEDIFF(day, '20160212', '20170212') computes the difference in days between February 12, 2016 and February 12, 2017, returning the value 366. Note that this function looks only at the parts from the requested one and above in the date and time hierarchy—not below. For example, the expression DATEDIFF(year, '20161231', '20170101') looks only at the year part, and hence returns 1. It doesn't look at the month and day parts of the values.

The DATEDIFF function returns a value of an INT type. If the difference doesn't fit in a four-byte integer, use the DATEDIFF_BIG function instead. This function returns the result as a BIGINT type.

Offset

T-SQL supports three functions related to date and time values with an offset: SWITCHOFFSET, TODATETIMEOFFSET, and AT TIME ZONE.

The SWITCHOFFSET function returns an input DATETIMEOFFSET value adjusted to a requested target offset (from the UTC time zone). For example, consider the expression SWITCHOFFSET(SYSDATETIMEOFFSET(), '-08:00'). Regardless of the offset of the SQL Server instance you are connected to, you request to present the current date and time value in terms of offset '-08:00'. If the system's offset is, say, '-05:00', the function will compensate for this by subtracting three hours from the input value.

The TODATETIMEOFFSET function is used for a different purpose. You use it to construct a DATETIMEOFFSET value from two inputs: the first is a date and time value that is not offset-aware, and the second is the offset. You can use this function to convert a value that is not offset aware to a target offset typed value without the need to manually convert the value and the offset to character strings with the right style and then to DATETIMEOFFSET. You can also use this function when migrating from data that is not offset-aware, where you keep the local date and time value in one attribute, and the offset in another, to offset-aware data. Say you have the local date and time in an attribute called mydatetime, and the offset in an attribute called theoffset. You add an attribute called mydatetimeoffset of a DATETIMEOFFSET type to the table. You then update the new attribute to the expression TODATETIMEOFFSET(mydatetime, theoffset), and then drop the original attributes mydatetime and theoffset from the table.

The following code demonstrates using both functions:

```
SELECT
  SWITCHOFFSET('20170212 14:00:00.0000000 -05:00', '-08:00') AS [SWITCHOFFSET],
  TODATETIMEOFFSET('20170212 14:00:00.0000000', '-08:00') AS [TODATETIMEOFFSET];
```

This code generates the following output:

```
SWITCHOFFSET                      TODATETIMEOFFSET
--------------------------------- ---------------------------------
2017-02-12 11:00:00.0000000 -08:00 2017-02-12 14:00:00.0000000 -08:00
```

What's tricky about both functions is that many time zones support a daylight savings concept where twice a year you move the clock by an hour. So when you capture the date and time value, you need to make sure that you also capture the right offset depending on whether it's currently daylight savings or not. For instance, in the time zone Pacific Standard Time the offset from UTC is '-07:00' when it's daylight savings time and '-08:00' when it isn't.

T-SQL supports a function called AT TIME ZONE that can be used instead of both the SWITCHOFFSET and the TODATETIMEOFFSET functions, and that uses named time zones instead of offsets. This way you don't need to worry about capturing the right offset and whether it's daylight savings time or not, you just capture the time zone name. When the input value is of a DATETIMEOFFSET type, the function assumes that you want to treat the conversion similar to SWITCHOFFSET. For instance, never mind what's the current time zone

setting in the current instance, suppose that you want to return now as a DATETIMEOFFSET value in the time zone Pacific Standard Time. You use the expression: SYSDATETIMEOFFSET() AT TIME ZONE 'Pacific Standard Time'. If when you're running this code it's currently daylight savings, the function will switch the input value to offset '-07:00', otherwise to '-08:00'.

When the input value is not an offset-aware value, the AT TIME ZONE function assumes that you want to treat the conversion similar to TODATETIMEOFFSET and that the source value is already of the target time zone. Again, you don't need to worry about daylight savings considerations. Based on the point in the year, the function will know whether to apply daylight savings time. Here's an example demonstrating the use of the function with an input that is not offset-aware:

```
DECLARE @dt AS DATETIME2 = '20170212 14:00:00.0000000';
SELECT @dt AT TIME ZONE 'Pacific Standard Time';
```

This code generates the following output:

```
-------------------------------
2017-02-12 14:00:00.0000000 -08:00
```

The get the set of supported time zones query the view sys.time_zone_info.

When you store date and time values as a type that is not offset aware and you can present them as DATETIMEOFFSET values of a different target time zone, you need to apply the AT TIME ZONE function twice—once to convert the value to DATETIMEOFFSET with the source time zone and another to switch the now DATETIMEOFFSET value from its current time zone to the target one. For instance, suppose that you have a column called lastmodified that is typed as DATETIME2 and holds the value in UTC terms. You want to present it in the time zone Pacific Standard Time. You use the following expression: lastmodified AT TIME ZONE 'UTC' AT TIME ZONE 'Pacific Standard Time.'

Character functions

T-SQL was not really designed to support very sophisticated character string manipulation functions, so you won't find a very large set of such functions. This section describes the character string functions that T-SQL does support, arranged in categories.

Concatenation

Character string concatenation is a very common need. T-SQL supports two ways to concatenate strings—one with the plus (+) operator, and another with the CONCAT function.

Here's an example for concatenating strings in a query by using the + operator:

```
SELECT empid, country, region, city,
  country + N', ' + region + N', ' + city AS location
FROM HR.Employees;
```

This query generates the following output:

```
empid       country         region          city            location
----------- --------------- --------------- --------------- ----------------
1           USA             WA              Seattle         USA, WA, Seattle
2           USA             WA              Tacoma          USA, WA, Tacoma
3           USA             WA              Kirkland        USA, WA, Kirkland
4           USA             WA              Redmond         USA, WA, Redmond
5           UK              NULL            London          NULL
6           UK              NULL            London          NULL
7           UK              NULL            London          NULL
8           USA             WA              Seattle         USA, WA, Seattle
9           UK              NULL            London          NULL
```

Observe that when any of the inputs is NULL, the + operator returns a NULL. That's standard behavior that can be changed by turning off a session option called CONCAT_NULL_YIELDS_NULL, though it's not recommended to rely on such nonstandard behavior. If you want to substitute a NULL with an empty string, there are a number of ways for you to do this programmatically. One option is to use ISNULL or COALESCE functions to replace a NULL with an empty string. For example, in this data, only region can be NULL, so you can use the following query to replace a comma plus region with an empty string when region is NULL:

```
SELECT empid, country, region, city,
  country + ISNULL(N', ' + region, N'') + N', ' + city AS location
FROM HR.Employees;
```

Another option is to use the CONCAT function which, unlike the + operator, substitutes a NULL input with an empty string. Here's how the query looks:

```
SELECT empid, country, region, city,
  CONCAT(country, N', ' + region, N', ' + city) AS location
FROM HR.Employees;
```

Here's the output of this query:

```
empid       country         region          city            location
----------- --------------- --------------- --------------- ----------------
1           USA             WA              Seattle         USA, WA, Seattle
2           USA             WA              Tacoma          USA, WA, Tacoma
3           USA             WA              Kirkland        USA, WA, Kirkland
4           USA             WA              Redmond         USA, WA, Redmond
5           UK              NULL            London          UK, London
6           UK              NULL            London          UK, London
7           UK              NULL            London          UK, London
8           USA             WA              Seattle         USA, WA, Seattle
9           UK              NULL            London          UK, London
```

Observe that this time, when region was NULL, it was replaced with an empty string.

Substring extraction and position

This section covers functions that you can use to extract a substring from a string, and identify the position of a substring within a string.

With the SUBSTRING function, you can extract a substring from a string given as the first argument, starting with the position given as the second argument, and a length given as the third argument. For example, the expression SUBSTRING('abcde', 1, 3) returns 'abc'. If the third argument is greater than what would get you to the end of the string, the function doesn't fail; instead, it just extracts the substring until the end of the string.

The LEFT and RIGHT functions extract a requested number of characters from the left and right ends of the input string, respectively. For example, LEFT('abcde', 3) returns 'abc' and RIGHT('abcde', 3) returns 'cde'.

The CHARINDEX function returns the position of the first occurrence of the string provided as the first argument within the string provided as the second argument. For example, the expression CHARINDEX(' ','Inigo Montoya') looks for the first occurrence of a space in the second input, returning 6 in this example. Note that you can provide a third argument indicating to the function the position where to start looking.

You can combine, or nest, functions in the same expression. For example, suppose you query a table with an attribute called fullname formatted as '<first> <last>', and you need to write an expression that extracts the first name part. You can use the following expression:

```
LEFT(fullname, CHARINDEX(' ', fullname) - 1)
```

T-SQL also supports a function called PATINDEX that, like CHARINDEX, you can use to locate the first position of a string within another string. But whereas with CHARINDEX you're looking for a constant string, with PATINDEX you're looking for a pattern. The pattern is formed very similar to the LIKE patterns that you're probably familiar with, where you use wildcards like % for any string, _ for a single character, and square brackets ([]) representing a single character from a certain list or range. If you're not familiar with such pattern construction, see the topics "PATINDEX (Transact-SQL)" and "LIKE (Transact-SQL)" in the T-SQL documentation at *https://msdn.microsoft.com/en-us/library/ms188395.aspx* and *https://msdn.microsoft.com/en-us/library/ms179859.aspx*. As an example, the expression PATINDEX('%[0-9]%', 'abcd123efgh') looks for the first occurrence of a digit (a character in the range 0–9) in the second input, returning the position 5 in this case.

String length

T-SQL provides two functions that you can use to measure the length of an input value—LEN and DATALENGTH.

The LEN function returns the length of an input string in terms of the number of characters. Note that it returns the number of characters, not bytes, whether the input is a regular character or Unicode character string. For example, the expression LEN(N'xyz') returns 3. If there are any trailing spaces, LEN removes them.

The DATALENGTH function returns the length of the input in terms of number of bytes. This means, for example, that if the input is a Unicode character string, it will count 2 bytes per character. For example, the expression DATALENGTH(N'xyz') returns 6. Note also that, unlike LEN, the DATALENGTH function doesn't remove trailing spaces.

String alteration

T-SQL supports a number of functions that you can use to apply alterations to an input string. Those are REPLACE, REPLICATE, and STUFF.

With the REPLACE function, you can replace in an input string provided as the first argument all occurrences of the string provided as the second argument, with the string provided as the third argument. For example, the expression REPLACE('.1.2.3.', '.', '/') substitutes all occurrences of a dot (.) with a slash (/), returning the string '/1/2/3/'.

The REPLICATE function allows you to replicate an input string a requested number of times. For example, the expression REPLICATE('0', 10) replicates the string '0' ten times, returning '0000000000'.

The STUFF function operates on an input string provided as the first argument; then, from the character position indicated as the second argument, deletes the number of characters indicated by the third argument. Then it inserts in that position the string specified as the fourth argument. For example, the expression STUFF(',x,y,z', 1, 1, '') removes the first character from the input string, returning 'x,y,z'.

Formatting

This section covers functions that you can use to apply formatting options to an input string. Those are the UPPER, LOWER, LTRIM, RTRIM, and FORMAT functions.

The first four functions are self-explanatory (uppercase form of the input, lowercase form of the input, input after removal of leading spaces, and input after removal of trailing spaces). Note that there's no TRIM function that removes both leading and trailing spaces; to achieve this, you need to nest one function call within another, as in RTRIM(LTRIM(<input>)).

As mentioned earlier, with the FORMAT function, you can format an input value based on a .NET format string. I demonstrated an example with date and time values. As another example, this time with numeric values, the expression FORMAT(1759, '0000000000') formats the input number as a character string with a fixed size of 10 characters with leading zeros, returning '0000001759'. The same thing can be achieved with the format string 'd10', meaning decimal value with 10 digits, with the expression FORMAT(1759, 'd10').

String splitting

T-SQL supports a table-valued function called STRING_SPLIT that accepts a charter string with a separated list of values provided as the first input, and a character string with the separator as the second input, and returns a result set with a column called value holding the individual split strings. The function supports all character string types for both inputs—regular and

Unicode. The type of the result value column, which is actually named *value*, is NVARCHAR if the first input is of a Unicode character string type, and VARCHAR otherwise.

As an example, the following code splits an input string that holds a separated list of order IDs:

```
DECLARE @orderids AS VARCHAR(MAX) = N'10248,10542,10731,10765,10812';

SELECT value
FROM STRING_SPLIT(@orderids, ',');
```

This code generates the following output:

```
Value
------
10248
10542
10731
10765
10812
```

Suppose that @orderids is a parameter provided to a stored procedure or a function, and that routine is supposed to split those IDs, and join the result with the Sales.Orders table to return information about the input orders. You achieve this with the following query, using a local variable here for simplicity:

```
DECLARE @orderids AS VARCHAR(MAX) = N'10248,10542,10731,10765,10812';

SELECT O.orderid, O.orderdate, O.custid, O.empid
FROM STRING_SPLIT(@orderids, ',') AS K
  INNER JOIN Sales.Orders AS O
    ON O.orderid = CAST(K.value AS INT);
```

This query generates the following output:

```
orderid     orderdate  custid      empid
----------- ---------- ----------- -----------
10248       2014-07-04 85          5
10542       2015-05-20 39          1
10812       2016-01-02 66          5
10765       2015-12-04 63          3
10731       2015-11-06 14          7
```

CASE expressions and related functions

T-SQL supports an expression called CASE and a number of related functions that you can use to apply conditional logic to determine the returned value. Many people incorrectly refer to CASE as a statement. A statement performs some kind of an action or controls the flow of the code, and that's not what CASE does; CASE returns a value, and hence is an expression.

The CASE expression has two forms—the simple form and the searched form. Here's an example of the simple CASE form issued against the sample database TSQLV4.

```
SELECT productid, productname, unitprice, discontinued,
  CASE discontinued
    WHEN 0 THEN 'No'
    WHEN 1 THEN 'Yes'
    ELSE 'Unknown'
  END AS discontinued_desc
FROM Production.Products;
```

The simple form compares an input expression (in this case the attribute discontinued) to multiple possible scalar when expressions (in this case, 0 and 1), and returns the result expression (in this case, 'No' and 'Yes', respectively) associated with the first match. If there's no match and an ELSE clause is specified, the else expression (in this case, 'Unknown') is returned. If there's no ELSE clause, the default is ELSE NULL. Here's an abbreviated form of the output of this query:

```
productid   productname    unitprice             discontinued discontinued_desc
----------- -------------- --------------------- ------------ -----------------
1           Product HHYDP  18.00                 0            No
2           Product RECZE  19.00                 0            No
3           Product IMEHJ  10.00                 0            No
4           Product KSBRM  22.00                 0            No
5           Product EPEIM  21.35                 1            Yes
6           Product VAIIV  25.00                 0            No
...
```

The searched form of the CASE expression is more flexible. Instead of comparing an input expression to multiple possible expressions, it uses predicates in the WHEN clauses, and the first predicate that evaluates to true determines which when expression is returned. If none is true, the CASE expression returns the else expression. Here's an example:

```
SELECT productid, productname, unitprice,
  CASE
    WHEN unitprice < 20.00 THEN 'Low'
    WHEN unitprice < 40.00 THEN 'Medium'
    WHEN unitprice >= 40.00 THEN 'High'
    ELSE 'Unknown'
  END AS pricerange
FROM Production.Products;
```

In this example, the CASE expression returns a description of the product's unit price range. When the unit price is below $20.00, it returns 'Low', when it's $20.00 or more and below $40.00, it returns 'Medium', and when it's $40.00 or more, it returns 'High'. There's an ELSE clause for safety; if the input is NULL, the else expression returned is 'Unknown'. Notice that the second when predicate didn't need to check whether the value is $20.00 or more explicitly. That's because the when predicates are evaluated in order and the first when predicate did not evaluate to true. Here's an abbreviated form of the output of this query.

```
productid   productname    unitprice             pricerange
----------- -------------- --------------------- ----------
1           Product HHYDP  18.00                 Low
2           Product RECZE  19.00                 Low
3           Product IMEHJ  10.00                 Low
4           Product KSBRM  22.00                 Medium
5           Product EPEIM  21.35                 Medium
...
```

T-SQL supports a number of functions that can be considered as abbreviates of the CASE expression. Those are the standard COALESCE and NULLIF functions, and the nonstandard ISNULL, IIF, and CHOOSE.

The COALESCE function accepts a list of expressions as input and returns the first that is not NULL, or NULL if all are NULLs. If all inputs are the untyped NULL constant, as in COALESCE(NULL, NULL, NULL), SQL Server generates an error. For example, the expression COALESCE(NULL, 'x', 'y') returns 'x'. More generally, the expression:

```
COALESCE(<exp1>, <exp2>, ..., <expn>)
```

is similar to the following:

```
CASE
  WHEN <exp1> IS NOT NULL THEN <exp1>
  WHEN <exp2> IS NOT NULL THEN <exp2>
  ...
  WHEN <expn> IS NOT NULL THEN <expn>
  ELSE NULL
END
```

A typical use of COALESCE is to substitute a NULL with something else. For example, the expression COALESCE(region, '') returns region if it's not NULL and returns an empty string if it is NULL.

T-SQL supports a nonstandard function called ISNULL that is similar to the standard COALESCE, but it's a bit more limited in the sense that it supports only two inputs. Like COALESCE, it returns the first input that is not NULL. So, instead of COALESCE(region, ''), you could use ISNULL(region, ''). If there's a requirement to use standard code in the application when possible, you should prefer COALESCE in such a case.

There are a few interesting differences between COALESCE and ISNULL that you should be aware of when working with these functions. One is which input determines the type of the output. Consider the following code:

```
DECLARE
  @x AS VARCHAR(3) = NULL,
  @y AS VARCHAR(10) = '1234567890';

SELECT COALESCE(@x, @y) AS [COALESCE], ISNULL(@x, @y) AS [ISNULL];
```

Here's the output of this code:

```
COALESCE   ISNULL
---------- ------
1234567890 123
```

Observe that the type of the COALESCE expression is determined by the returned element, whereas the type of the ISNULL expression is determined by the first input.

The other difference is when you use the SELECT INTO statement, which writes a query result into a target table, including creating the table. Suppose the SELECT list of a SELECT INTO statement contains the expressions COALESCE(col1, 0) AS newcol1 versus ISNULL(col1, 0) AS newcol1. If the source attribute col1 is defined as NOT NULL, both expressions will produce an attribute in the result table defined as NOT NULL. However, if the source attribute col1 is defined as allowing NULLs, COALESCE will create a result attribute allowing NULLs, whereas ISNULL will create one that disallows NULLs.

T-SQL also supports the standard NULLIF function. This function accepts two input expressions, returns NULL if they are equal, and returns the first input if they are not. For example, consider the expression NULLIF(col1, col2). If col1 is equal to col2, the function returns a NULL; otherwise, it returns the col1 value.

As for IIF and CHOOSE, these are nonstandard T-SQL functions that were added to simplify migrations from Microsoft Access platforms. Because these functions aren't standard and there are simple standard alternatives with CASE expressions, it is not usually recommended that you use them. However, when you are migrating from Access to SQL Server, these functions can help with smoother migration, and then gradually you can refactor your code to use the available standard functions. With the IIF function, you can return one value if an input predicate is true and another value otherwise. The function has the following form:

```
IIF(<predicate>, <true_result>, <false_or_unknown_result>)
```

This expression is equivalent to the following:

```
CASE WHEN <predicate> THEN <true_result> ELSE <false_or_unknown_result> END
```

For example, the expression IIF(orderyear = 2017, qty, 0) returns the value in the qty attribute when the orderyear attribute is equal to 2017, and zero otherwise.

The CHOOSE function allows you to provide a position and a list of expressions, and returns the expression in the indicated position. The function takes the following form:

```
CHOOSE(<pos>, <exp1>, <exp2>, …, <expn>)
```

For example, the expression CHOOSE(2, 'x', 'y', 'z') returns 'y'. Again, it's straightforward to replace a CHOOSE expression with a logically equivalent CASE expression; but the point in supporting CHOOSE, as well as IIF, is to simplify migrations from Access to SQL Server as a temporary solution.

System functions

System functions return information about various aspects of the system. Here I highlight a few of the functions. You can find the full list in the Transact-SQL documentation at *https://msdn.microsoft.com/en-us/library/ms187786.aspx*.

The @@ROWCOUNT and ROWCOUNT_BIG functions

The @@ROWCOUNT function is a very popular function that returns the number of rows affected by the last statement that you executed. It's very common to use it to check if the previous statement affected any rows by checking that the function's result is zero or greater than zero. For example, the following code queries the input employee row, and prints a message if the requested employee was not found:

```
DECLARE @empid AS INT = 10;

SELECT empid, firstname, lastname
FROM HR.Employees
WHERE empid = @empid;

IF @@ROWCOUNT = 0
  PRINT CONCAT('Employee ', CAST(@empid AS VARCHAR(10)), ' was not found.');
```

This code generates the following output:

```
empid       firstname  lastname
----------- ---------- --------------------

(0 row(s) affected)

Employee 10 was not found.
```

The @@ROWCOUNT function returns an INT typed value. If the row count can exceed the maximum INT value (2,147,483,647), use the ROWCOUNT_BIG function, which returns a BIGINT typed value.

Compression functions

T-SQL supports a function called COMPRESS that enables you to compress an input character or binary string using the GZIP algorithm into a result binary string. It also supports a function called DECOMPRESS that allows you to decompress a previously compressed string.

Note that you need to explicitly invoke the COMPRESS function to compress the input string before you store the result compressed binary string in a table. For example, supposed that you have a stored procedure parameter called @notes of the type NVARCHAR(MAX) that you need to compress and store the result in a table, in a column called notes. As part of your INSERT statement against the target table, the VALUES clause includes the expression COMPRESS(@notes) as the target value for the target column. Your code might look like this:

```
INSERT INTO dbo.MyNotes(notes)
  VALUES(COMPRESS(@notes));
```

When you later query the table, you use the expression DECOMPRESS(notes) to decompress the column value. However, because the result is a binary string, you need to convert it to the target type using the expression CAST(DECOMPRESS(notes) AS NVARCHAR(MAX)). Your code might look like this.

```
SELECT keycol
  CAST(DECOMPRESS(notes) AS NVARCHAR(MAX)) AS notes
FROM dbo.MyNotes;
```

Context info and session context

When you need to pass information from one level in the call stack to another, you usually use parameters. For instance, if you want to pass something to a procedure, you use an input parameter, and if you want to return something back, you use an output parameter. However, certain modules in T-SQL, for example, triggers, are by design *niladic*, meaning they don't support parameters. One technique to pass information between an outer level and a niladic module is to use either context info or session context.

Context info is a binary string of up to 128 bytes that is associated with your session. You write to it using the SET CONTEXT_INFO command and read it using the CONTEXT_INFO function. For example, the following code writes the value 'us_english,' after converting it to a binary string, as the current session's context info:

```
DECLARE @mycontextinfo AS VARBINARY(128) = CAST('us_english' AS VARBINARY(128));
SET CONTEXT_INFO @mycontextinfo;
```

You can read the context info from anywhere in your session, including triggers as follows:

```
SELECT CAST(CONTEXT_INFO() AS VARCHAR(128)) AS mycontextinfo;
```

This code generates the following output:

```
Mycontextinfo
--------------
us_english
```

The tricky thing about context info is that there's only one such binary string for the session. If you need to use it to store multiple values from different places in the code, you need to designate different parts of it for the different values. Every time you need to store a value, you need to read the current contents, and reconstruct it with the new value planted in the right section, being careful not to overwrite existing used parts. The potential to corrupt meaningful information is high.

T-SQL provides a tool called *session context* as a more convenient and robust alternative to context info. With session context, you store key-value pairs, where the key is a name of a sysname type (internally mapped to NVARCHAR(128)) that you assign to your session's variable, and the value is a SQL_VARIANT typed value that is associated with the key. You can also mark the pair as read only, and then until the session resets, no one will be able to overwrite the value associated with that key. You create the key and set its associated value using the sp_set_session_context stored procedure and read it using the SESSION_CONTEXT function.

As an example, the following code creates a key called language and associates with it the value 'us_english', marking it as read only.

```
EXEC sys.sp_set_session_context
  @key = N'language', @value = 'us_english', @read_only = 1;
```

Then when you need to read the value from anywhere in your session, you use the following code:

```
SELECT SESSION_CONTEXT(N'language') AS [language];
```

This code generates the following output:

```
Language
-----------
us_english
```

GUID and identity functions

T-SQL provides a number of solutions for generating values that you can use as keys for your rows. T-SQL also provides system functions to generate and query the newly generated keys.

If you need to generate a key that is globally unique, even across systems, you use the NEWID function to generate it as a UNIQUEIDENTIFER typed value. As an example, run the following code:

```
SELECT NEWID() AS myguid;
```

You can run this code several times and see that every time you get a different globally unique identifier (GUID). For instance, in one of my executions of this code I got the following output:

```
Myguid
------------------------------------
203B8382-77E4-4B7E-B6B9-260CC7A9CB8C
```

If you want the GUIDs to always increase within the machine, use the NEWSEQUENTIALID system function instead. Note that you cannot invoke this function independently, rather only as an expression in a default constraint that is associated with a column.

If you need a numeric key generator, you use either a sequence object or the identity column property. The former is an independent object in the database that you create using the CREATE SEQUENCE command. Once created, every time you need a new value, you invoke the function NEXT VALUE FOR <sequence_name>. The latter is a property of a column in a table. SQL Server generates a new key only as part of an INSERT statement that you submit against the target table, where you ignore the column with the identity property. After adding the row, you query the system function SCOPE_IDENTITY to get the last identity value that was generated in the same session and scope. In *the same scope* I mean that if a trigger was fired and also added a row to a table with an identity property, this will not affect the value that the function will return. If you want to get the last identity value generated in your session, irrespective of scope, you query the system function @@IDENTITY. You can find examples for using both the identity property and the sequence object later in this chapter in Skill 1.4.

Arithmetic operators and aggregate functions

T-SQL supports the four classic arithmetic operators + (add), - (subtract), * (multiply), / (divide), as well as the fifth operator % (modulo). The last computes the remainder of an integer division. T-SQL also supports aggregate functions, which you apply to a set of rows, and get a single value back.

Arithmetic operators

For the most part, work with these arithmetic operators is intuitive. They follow classic arithmetic operator precedence rules, which say that multiplication, division and modulo precede addition and subtraction. To change precedence of operations, use parentheses because they precede arithmetic operators. For example, consider the following expression:

```
SELECT 2 + 3 * 2 + 10 / 2;
```

It is equivalent to the following expression:

```
SELECT 2 + (3 * 2) + (10 / 2);
```

The result of this expression is 13.

If you want to evaluate the operations from left to right, you need to use parentheses as follows:

```
SELECT ((2 + 3) * 2 + 10) / 2;
```

This expression evaluates to 10.

The data types of the operands in an arithmetic computation determine the data type of the result. If the operands are integers, the result of arithmetic operations is an integer. With this in mind, consider the following expression:

```
SELECT 9 / 2;
```

With integer division, the result of this expression is 4 and not 4.5. Obviously, when using constants, you can simply specify numeric values instead of integer values to get numeric division; however, when the operands are integer columns or parameters, but you need numeric division, you have two options. One option is to explicitly cast the operands to a numeric type with the appropriate precision and scale as follows:

```
DECLARE @p1 AS INT = 9, @p2 AS INT = 2;
SELECT CAST(@p1 AS NUMERIC(12, 2)) / CAST(@p2 AS NUMERIC(12, 2));
```

The rules for determining the precision and scale of the result of the computation can be found at *https://msdn.microsoft.com/en-us/library/ms190476.aspx*. The result of this expression is 4.500000000000000. The operation here is division. The applicable formula to calculate the precision here is p1 - s1 + s2 + max(6, s1 + p2 + 1), which when applied to our inputs results in 27. The formula for the scale is max(6, s1 + p2 + 1), which in this case results in 15.

Another option is to multiply the first operand by a numeric constant, and this way force implicit conversion of both the first and the second operands to a numeric type as follows:

```
DECLARE @p1 AS INT = 9, @p2 AS INT = 2;
SELECT 1.0 * @p1 / @p2;
```

Aggregate functions

An aggregate function is a function that you apply to a set of rows and get a single value back. T-SQL supports aggregate functions such as SUM, COUNT, MIN, MAX, AVG and others. You can find the full list at *https://msdn.microsoft.com/en-us/library/ms173454.aspx*.

Aggregate functions ignore NULL inputs when applied to an expression. The COUNT(*) aggregate just counts rows, and returns the result as an INT value. Use COUNT_BIG to return the row count as a BIGINT value. If you want to apply an aggregate function to distinct values, add the DISTINCT clause, as in COUNT(DISTINCT custid).

You can apply aggregate functions in explicit grouped queries as the following example shows:

```
SELECT empid, SUM(qty) AS totalqty
FROM Sales.OrderValues
GROUP BY empid;
```

In a grouped query the aggregate is applied per group, and returns a single value per group, as part of the single result row that represents the group. This query generates the following output:

```
empid       totalqty
----------- -----------
9           2670
3           7852
6           3527
7           4654
1           7812
4           9798
5           3036
2           6055
8           5913
```

An aggregate function can also be applied as a scalar aggregate in an implied grouped query. The presence of the aggregate function causes the query to be considered a grouped one, as in the following example:

```
SELECT SUM(qty) AS totalqty FROM Sales.OrderValues;
```

This query returns the grand total quantity 51,317.

Like with arithmetic operators, also with aggregate functions like AVG, the data type of the input determines the data type of the result. For instance, the following query produces an integer average:

```
SELECT AVG(qty) AS avgqty FROM Sales.OrderValues;
```

The result of this average is the integer 61.

You can use the two aforementioned options that I described for arithmetic operations to get a numeric average. Either explicitly cast the input to a numeric type as follows:

```
SELECT AVG(CAST(qty AS NUMERIC(12, 2))) AS avgqty FROM Sales.OrderValues;
```

Or implicitly as follows:

```
SELECT AVG(1.0 * qty) AS avgqty FROM Sales.OrderValues;
```

This time you get the result 61.827710.

If you're wondering why the scale of the result value here is 6 digits, the AVG function is handled internally as a sum divided by a count. The scale of the input expression (1.0 * qty) is the sum of the scales of the operands (1 for 1.0 and 0 for the integer qty), which in our case is 1. The sum aggregate's scale is the maximum scale among the input values, which in our case is 1. Then the scale of the result of the division between the sum and the count is based on the formula max(6, s1 + p2 + 1), which in our case is 6.

As an alternative to grouping, aggregate functions can be applied in windowed queries, as window aggregates. This, as well as further aspects of grouping and aggregation are covered in Chapter 2, Skill 2.3.

Example involving arithmetic operators and aggregate functions

As mentioned, the % (modulo) operator computes the remainder of an integer division. Suppose that you were tasked with computing the median quantity (qty column) from the Sales.OrderValues view using a continuous distribution model. This means that if there are an odd number of rows you need to return the middle quantity, and if there are an even number of rows, you need to return the average of the two middle quantities. Using the COUNT aggregate, you can first count how many rows there are and store in a variable called @cnt. Then you can compute parameters for the OFFSET-FETCH filter to specify, based on qty ordering, how many rows to skip (offset value) and how many to filter (fetch value). The number of rows to skip is (@cnt - 1) / 2. It's clear that for an odd count this calculation is correct because you first subtract 1 for the single middle value, before you divide by 2.

This also works correctly for an even count because the division used in the expression is integer division; so, when subtracting 1 from an even count, you're left with an odd value. When dividing that odd value by 2, the fraction part of the result (.5) is truncated. The number of rows to fetch is 2 - (@cnt % 2). The idea is that when the count is odd the result of the modulo operation is 1, and you need to fetch 1 row. When the count is even the result of the modulo operation is 0, and you need to fetch 2 rows. By subtracting the 1 or 0 result of the modulo operation from 2, you get the desired 1 or 2, respectively. Finally, to compute the median quantity, take the one or two result quantities, and apply an average after converting the input integer value to a numeric one as follows:

```
DECLARE @cnt AS INT = (SELECT COUNT(*) FROM Sales.OrderValues);

SELECT AVG(1.0 * qty) AS median
```

```
FROM ( SELECT qty
       FROM Sales.OrderValues
       ORDER BY qty
       OFFSET (@cnt - 1) / 2 ROWS FETCH NEXT 2 - @cnt % 2 ROWS ONLY ) AS D;
```

You cannot apply the AVG aggregate directly in the query with the OFFSET-FETCH filter. That's because if you did, there would have been implied grouping, which happens in the third step of logical query processing. Then the reference to the detail qty column in the ORDER BY clause, which is processed in the sixth logical query processing step, would have been invalid. Therefore, the solution defines a derived table (a table subquery in the FROM clause) called D that represents the one or two quantities that need to participate in the median calculation, and then the outer query handles the average calculation. This query returns the median quantity 50.000000.

Search arguments

One of the most important aspects of query tuning to know is what a search argument is. A *search argument*, or *SARG* in short, is a filter predicate that enables the optimizer to rely on index order. The filter predicate uses the following form (or a variant with two delimiters of a range, or with the operand positions flipped):

```
WHERE <column> <operator> <expression>
```

Such a filter is sargable if:

1. You don't apply manipulation to the filtered column.
2. The operator identifies a consecutive range of qualifying rows in the index. That's the case with operators like =, >, >=, <, <=, BETWEEN, LIKE with a known prefix, and so on. That's not the case with operators like <>, LIKE with a wildcard as a prefix.

In most cases, when you apply manipulation to the filtered column, the optimizer doesn't try to be too smart and understand the meaning of the calculation, and if index ordering can still be relied on. It simply assumes that the result values might sort differently than the source values, and therefore index ordering can't be trusted. This, for example, can prevent the ability to rely on the index to filter the data by applying a seek and range scan.

As an example, consider the following query:

```
SELECT orderid, orderdate
FROM Sales.Orders
WHERE YEAR(orderdate) = 2015;
```

The plan for this query is shown in Figure 1-15.

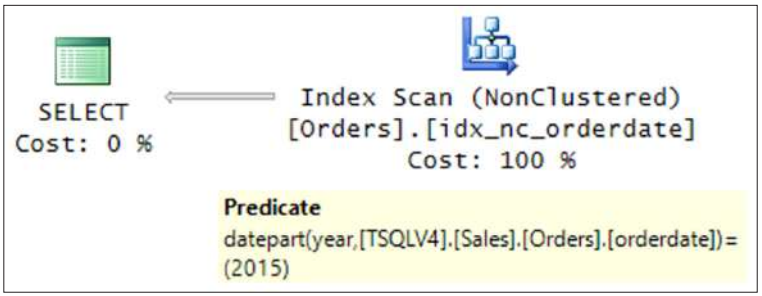


FIGURE 1-15 Plan with an index scan for example with date range

There is a nonclustered covering index defined on the orderdate column as the leading key, with orderid being implicitly part of the index because it's the clustered index key. Clearly, the qualifying rows appear in a consecutive range in the index. However, the manipulation applied to the orderdate column with the YEAR function prevents the filter's sargability, and causes the optimizer to scan the whole index. The predicate is applied as a residual predicate, and shows up under the scan's Predicate property.

The alternative that is considered a search argument expresses the predicate as a range, without applying manipulation to the filtered column as follows:

```
SELECT orderid, orderdate
FROM Sales.Orders
WHERE orderdate >= '20150101'
  AND orderdate < '20160101';
```

The plan for this query is shown in Figure 1-16.

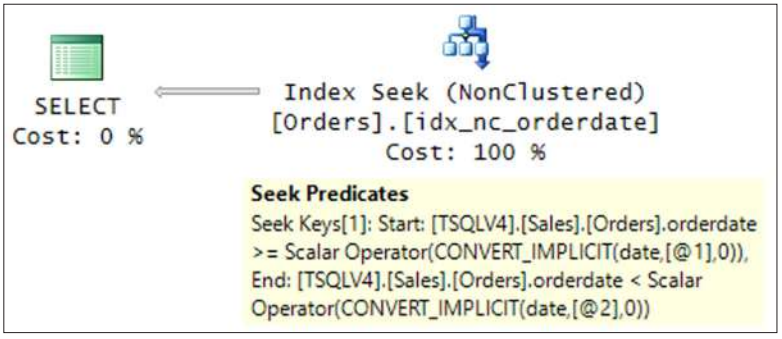


FIGURE 1-16 Plan with an index seek for example with date range

This time the plan applies a seek in the index and a range scan in the index leaf, physically touching only qualifying rows. The predicate appears under the Seek Predicates property.

Note that in some rare cases, Microsoft added logic to the optimizer to convert a nonsargable predicate to a sargable one. For instance, when using the predicate CAST(dt AS DATE) = '20170212', where dt is an indexed date and time typed column, SQL Server can compute two delimiters in an open-open interval, and then process the filter with a sargable predicate

where the column is greater than the first delimiter and less than the second. But because it's so uncommon for SQL Server to do such conversions, it's a best practice to make sure that you write in a sargable way to begin with.

There are plenty more typical examples where people fall into such traps. For instance, consider the following query:

```
DECLARE @todt AS DATE = '20151231';

SELECT orderid, orderdate
FROM Sales.Orders
WHERE DATEADD(day, -1, orderdate) < @todt;
```

This query resides in a stored procedure or user defined function that accepts a parameter called @todt (emulated here with an ad-hoc batch with a local variable), and is supposed to return all orders that were placed prior to the day after the input parameter. As written, the query isn't sargable due to the manipulated filtered column. The plan for this query is similar to the one shown earlier in Figure 1-15, only with the current query's filter predicate.

A simple mathematical transformation, applying a plus one day to the parameter instead of a minus one day to the column enables sargability as follows:

```
DECLARE @todt AS DATE = '20151231';

SELECT orderid, orderdate
FROM Sales.Orders
WHERE orderdate < DATEADD(day, 1, @todt);
```

The plan for this query is similar to the one shown earlier in Figure 1-16, only with the current query's filter predicate.

As another example, the following query returns employees with a last name that starts with the letter D:

```
SELECT empid, lastname
FROM HR.Employees
WHERE LEFT(lastname, 1) = N'D';
```

This filter isn't sargable due to the function LEFT that is applied to the lastname column, and therefore the plan for this query is similar to the one shown earlier in Figure 1-15, only with the index idx_nc_lastname and the current query's filter predicate.

The sargable alternative is to use the LIKE predicate as follows:

```
SELECT empid, lastname
FROM HR.Employees
WHERE lastname LIKE N'D%';
```

When using the LIKE predicate with a known prefix SQL Server internally translates the pattern to a closed-open interval, and process the filter as a range that is greater than or equal to the first delimiter and less than the second delimiter. The plan for this query is similar to the one shown earlier in Figure 1-16.

Yet another common example is when filtering a NULLable column. Consider the following query:

```
DECLARE @dt AS DATE = '20150212';

SELECT orderid, shippeddate
FROM Sales.Orders
WHERE shippeddate = @dt;
```

There's an index defined on the shippeddate column, with the orderid column implicitly included. Clearly, this filter is sargable, but there is a bug in this query. Unshipped orders are marked with a NULL in the shippeddate column, so when you want to see unshipped orders you pass a NULL as input as follows:

```
DECLARE @dt AS DATE = NULL;

SELECT orderid, shippeddate
FROM Sales.Orders
WHERE shippeddate = @dt;
```

This query uses an equality-based comparison, where a comparison between anything and a NULL, including between two NULLs, yields unknown, and therefore the result is an empty set.

One of the common techniques that people use to cope with such cases is to use the COALESCE or ISNULL function to replace a NULL in both sides with a value that can't normally appear in the data as follows:

```
DECLARE @dt AS DATE = NULL;

SELECT orderid, shippeddate
FROM Sales.Orders
WHERE ISNULL(shippeddate, '99991231') = ISNULL(@dt, '99991231');
```

You do get the correct result, but the filter isn't sargable, so the plan for this query is similar to the one shown earlier in Figure 1-15, only with the current index and query filter predicate.

One recommended solution that is considered sargable is to use the IS NULL predicate to check for NULLs as follows:

```
DECLARE @dt AS DATE = NULL;

SELECT orderid, shippeddate
FROM Sales.Orders
WHERE shippeddate = @dt
   OR (shippeddate IS NULL AND @dt IS NULL);
```

The plan for this query is similar to the one shown earlier in Figure 1-16.

This solution is correct, but if you have a conjunction of multiple predicates based on NULLable columns that you need to filter by, your WHERE clause will end up being long and convoluted. Recall the trick you used in the joins section when comparing NULLable columns using the EXISTS predicate and the INTERSECT set operator. You can apply the same trick here as follows:

```
DECLARE @dt AS DATE = NULL;

SELECT orderid, shippeddate
FROM Sales.Orders
WHERE EXISTS (SELECT shippeddate INTERSECT SELECT @dt);
```

With multiple columns you simply extend the SELECT lists in both sides. Set operators use a distinctness-based comparison and not an equality-based one, giving you the desired meaning without the need for special handling of NULLs. What's more, remarkably this form is sargable and therefore the plan for this query is also similar to the one shown earlier in Figure 1-16.

In the same way that manipulation of a filtered column prevents the sargability of a filter, such manipulation breaks the ordering property of the data. This means that even if there's an index on a column, SQL Server cannot rely on the index order to support an order-based algorithm for presentation ordering, window-function ordering, joining, grouping, distinctness, and so on.

Function determinism

Function determinism is a characteristic that indicates whether the function is guaranteed to return the same result given the same set of input values (including an empty set) in different invocations. If the function provides such a guarantee, it is said to be *deterministic*; otherwise, it is said to be *nondeterministic*. I am not going to go over the full list of functions and their determinism quality here. You can find the details at *https://msdn.microsoft.com/en-us/library/ms178091.aspx*. Here I am going to describe the different categories of determinism and illustrate each with a couple of examples. I also describe the limitations that nondeterministic functions impose.

There are three main categories of function determinism. There are functions that are always deterministic, those that are deterministic when invoked in a certain way, and those that are always nondeterministic.

Examples for functions that are always deterministic: all string functions, COALESCE, ISNULL, ABS, SQRT and many others. For instance, the expression ABS(-1759) always returns 1759.

Certain functions are either deterministic or not depending on how they're used. For example, the CAST function is not deterministic when converting from a character string to a date and time type or the other way around because the interpretation of the value might depend on the login's language. For instance, the expression CAST('02/12/17' AS DATE)

converts to February 12, 2017 under us_english, December 2, 2017 under British, and December 17, 2002 under Swedish and Japanese. The same applies to CONVERT when using certain styles. Another example is the RAND function. This function returns a float in the range 0 through 1. When using it with a seed, it is deterministic. For instance, run the following code several times:

```
SELECT RAND(1759);
```

You keep getting the same result, 0.746348756684839.

When you invoke the function without a seed, SQL Server computes a new seed based on the previous invocation and you get a pseudo random value. Run the following code several times and notice that you keep getting different results:

```
SELECT RAND();
```

In *pseudo*, I mean that even though technically without a seed the function is considered nondeterministic, if invoked right after an invocation with a seed, the result is repeatable. Run the following code several times:

```
SELECT RAND(1759);
SELECT RAND();
```

You repeatedly get the following output:

```
----------------------
0.746348756684839

----------------------
0.201391138037653
```

Certain functions are always nondeterministic, for example SYSDATETIME and NEWID. The former returns the current date and time value as a DATETIME2 typed value, and the latter returns a globally unique identifier as a UNIQUEIDENTIFIER typed value. NEWID returns a fairly random value, but its type is awkward to work with. To get a random integer value in a certain range, for instance, 1 through 10, use the following expression:

```
SELECT 1 + ABS(CHECKSUM(NEWID())) % 10;
```

By applying CHECKSUM to the result of NEWID you get a random integer. The absolute value modulo 10 gives you a random value in the range 0 through 9. Adding the result to 1 gives you a random value in the range 1 through 10.

Most nondeterministic functions are invoked once per query. That's the case for instance with SYSDATETIME and RAND (when invoked without a seed). The NEWID function is an exception in the sense that it gets invoked per row. Consider the following query:

```
SELECT empid, SYSDATETIME() AS dtnow, RAND() AS rnd, NEWID() AS newguid
FROM HR.Employees;
```

In one of the executions of this query on my system I got the following result (formatted as two outputs to fit on the page).

```
empid  dtnow                       rnd
------ --------------------------- ------------------
2      2016-10-02 09:35:46.8024874 0.980769010450262
7      2016-10-02 09:35:46.8024874 0.980769010450262
1      2016-10-02 09:35:46.8024874 0.980769010450262
5      2016-10-02 09:35:46.8024874 0.980769010450262
6      2016-10-02 09:35:46.8024874 0.980769010450262
8      2016-10-02 09:35:46.8024874 0.980769010450262
3      2016-10-02 09:35:46.8024874 0.980769010450262
9      2016-10-02 09:35:46.8024874 0.980769010450262
4      2016-10-02 09:35:46.8024874 0.980769010450262

empid  newguid
------ ------------------------------------
2      F2EC6CC7-E986-4A43-9ED9-1A08C56600D2
7      74A018CF-5E95-4C7F-BA8F-D707A3DD5177
1      B894EAAF-07C8-4CC1-AD20-4CE7DA4AFB81
5      10B86C1F-BD18-4E7E-8A70-195A239B54EA
6      20514C7A-4F3C-4C3F-BFBB-014E5FAC6B85
8      86C29355-16B2-4A48-9068-039ABBD56B85
3      D923AEBB-0FD7-424D-93C2-E68E199A24C1
9      E2940155-3C66-4F1A-BBF9-0D001AF9A4AB
4      2B3BE2DD-00CE-494E-B52C-989CA71A36E9
```

Keep this in mind, for instance, if you want to return the results ordered randomly, or select a random set of rows. For instance, suppose that you need to return a random set of three employees, and you use the following query in attempt to achieve this:

```
SELECT TOP (3) empid, firstname, lastname
FROM HR.Employees
ORDER BY RAND();
```

Run this code repeatedly and you probably keep getting the same result. Because the RAND function returns the same value in all rows, the ORDER BY is meaningless here. To get different random values in the different rows, order by NEWID, or for even better random distribution, apply CHECKSUM to NEWID as follows:

```
SELECT TOP (3) empid, firstname, lastname
FROM HR.Employees
ORDER BY CHECKSUM(NEWID());
```

Note that the use of a nondeterministic function in a computed column prevents the ability to create an index on the column. Similarly, the use of a nondeterministic function in a view prevents the ability to create a clustered index on the view. That's the case whether the function is always nondeterministic, or nondeterministic in certain cases.

Skill 1.4: Modify data

The T-SQL support for data manipulation language (DML) includes both statements that retrieve data (SELECT) and statements that modify data (INSERT, UPDATE, DELETE, TRUNCATE TABLE, and MERGE). The previous skills focused on data retrieval; this skill focuses on data modification.

This section covers how to:

- Write INSERT, UPDATE, and DELETE statements
- Determine which statements can be used to load data to a table based on its structure and constraints
- Construct Data Manipulation Language (DML) statements using the OUTPUT statement
- Determine the results of Data Definition Language (DDL) statements on supplied tables and data

Inserting data

T-SQL supports a number of different methods that you can use to insert data into your tables. Those include statements like INSERT VALUES, INSERT SELECT, INSERT EXEC, and SELECT INTO. This section covers these statements and demonstrates how to use them through examples.

Some of the code examples in this section use a table called Sales.MyOrders. Use the following code to create such a table in the sample database TSQLV4:

```
USE TSQLV4;
DROP TABLE IF EXISTS Sales.MyOrders;
GO

CREATE TABLE Sales.MyOrders
(
  orderid INT NOT NULL IDENTITY(1, 1)
    CONSTRAINT PK_MyOrders_orderid PRIMARY KEY,
  custid  INT NOT NULL,
  empid   INT NOT NULL,
  orderdate DATE NOT NULL
    CONSTRAINT DFT_MyOrders_orderdate DEFAULT (CAST(SYSDATETIME() AS DATE)),
  shipcountry NVARCHAR(15) NOT NULL,
  freight MONEY NOT NULL
);
```

Observe that the orderid column has an identity property defined with a seed 1 and an increment 1. This property generates the values in this column automatically when rows are inserted. As an alternative to the identity property you can use a sequence object to gener-

ate surrogate keys. For details about the sequence object and a comparison between the two options, see the following articles:

- Sequences part 1 at *http://sqlmag.com/sql-server/sequences-part-1*
- Sequences part 2 at *http://sqlmag.com/sql-server/sequences-part-2*
- Sequence and identity performance at *http://sqlmag.com/sql-server/sequence-and-identity-performance*

Also observe that the orderdate column has a default constraint with an expression that returns the current system's date.

INSERT VALUES

With the INSERT VALUES statement, you can insert one or more rows into a target table based on value expressions. Here's an example for a statement inserting one row into the Sales.MyOrderValues table:

```
INSERT INTO Sales.MyOrders(custid, empid, orderdate, shipcountry, freight)
  VALUES(2, 19, '20170620', N'USA', 30.00);
```

Specifying the target column names after the table name is optional but considered a best practice. That's because it enables you to control the source value to target column association, irrespective of the order in which the columns were defined in the table.

Without the target column list, you must specify the values in column definition order. If the underlying table definition changes but the INSERT statements aren't modified accordingly, this can result in either errors, or worse, values written to the wrong columns.

The INSERT VALUES statement does not specify a value for a column with an identity property because the property generates the value for the column automatically. Observe that the previous statement doesn't specify the orderid column. If you do want to provide your own value instead of letting the identity property do it for you, you need to first turn on a session option called IDENTITY_INSERT, as follows:

```
SET IDENTITY_INSERT <table> ON;
```

When you're done, you need to remember to turn it off.

Note that in order to use this option, you need quite strong permissions; you need to be the owner of the table or have ALTER permissions on the table.

Besides using the identity property, there are other ways for a column to get its value automatically in an INSERT statement. A column can have a default constraint associated with it like the orderdate column in the Sales.MyOrders table. If the INSERT statement doesn't specify a value for the column explicitly, SQL Server will use the default expression to generate that value. For example, the following statement doesn't specify a value for orderdate, and therefore SQL Server uses the default expression:

```
INSERT INTO Sales.MyOrders(custid, empid, shipcountry, freight)
  VALUES(3, 11, N'USA', 10.00);
```

Another way to achieve the same behavior is to specify the column name in the names list and the keyword DEFAULT in the respective element in the VALUES list. Here's an INSERT example demonstrating this:

```
INSERT INTO Sales.MyOrders(custid, empid, orderdate, shipcountry, freight)
  VALUES(3, 17, DEFAULT, N'USA', 30.00);
```

If you don't specify a value for a column, SQL Server first checks whether the column gets its value automatically—for example, from an identity property or a default constraint. If that's not the case, SQL Server checks whether the column allows NULLs, in which case it assumes a NULL. If that's not the case, SQL Server generates an error.

The INSERT VALUES statement doesn't limit you to inserting only one row; rather, it enables you to insert multiple rows. Simply separate the rows with commas, as follows:

```
INSERT INTO Sales.MyOrders(custid, empid, orderdate, shipcountry, freight) VALUES
  (2, 11, '20170620', N'USA', 50.00),
  (5, 13, '20170620', N'USA', 40.00),
  (7, 17, '20170620', N'USA', 45.00);
```

Note that the entire statement is considered one transaction, meaning that if any row fails to enter the target table, the entire statement fails and no row is inserted.

To see the result of running all INSERT examples in this section, query the table by using the following query:

```
SELECT * FROM Sales.MyOrders;
```

NOTE *SELECT **

As explained earlier, using SELECT * in production code is considered a bad practice. Here, SELECT * is used only for ad hoc querying purposes to examine the contents of tables after applying changes.

When I ran this code on my system, it returned the following output:

```
orderid  custid  empid  orderdate  shipcountry  freight
-------- ------- ------ ---------- ------------ --------
1        2       19     2017-06-20 USA          30.00
2        3       11     2017-02-12 USA          10.00
3        3       17     2017-02-12 USA          30.00
4        2       11     2017-06-20 USA          50.00
5        5       13     2017-06-20 USA          40.00
6        7       17     2017-06-20 USA          45.00
```

Remember that some of the INSERT examples relied on the default expression associated with the orderdate column, so naturally the dates you get reflect the date when you ran those examples.

INSERT SELECT

The INSERT SELECT statement inserts the result set returned by a query into the specified target table. As with INSERT VALUES, the INSERT SELECT statement supports optionally specifying the target column names. Also, you can omit columns that get their values automatically from an identity property, default constraint, or when allowing NULLs.

As an example, the following code inserts into the Sales.MyOrders table the result of a query against Sales.Orders returning orders shipped to customers in Norway:

```
SET IDENTITY_INSERT Sales.MyOrders ON;

INSERT INTO Sales.MyOrders(orderid, custid, empid, orderdate, shipcountry, freight)
  SELECT orderid, custid, empid, orderdate, shipcountry, freight
  FROM Sales.Orders
  WHERE shipcountry = N'Norway';

SET IDENTITY_INSERT Sales.MyOrders OFF;
```

The code turns on the IDENTITY_INSERT option against Sales.MyOrders in order to use the original order IDs and not let the identity property generate those.

Query the table after running this code:

```
SELECT * FROM Sales.MyOrders;
```

This query generates the following output (mostly new rows shown for brevity):

```
orderid  custid  empid  orderdate  shipcountry  freight
-------- ------- ------ ---------- ------------ --------
1        2       19     2017-06-20 USA          30.00
2        3       11     2017-02-12 USA          10.00
3        3       17     2017-02-12 USA          30.00
...
10387    70      1      2014-12-18 Norway       93.63
10520    70      7      2015-04-29 Norway       13.37
10639    70      7      2015-08-20 Norway       38.64
10831    70      3      2016-01-14 Norway       72.19
10909    70      1      2016-02-26 Norway       53.05
11015    70      2      2016-04-10 Norway       4.62
```

Setting IDENTITY_INSERT to OFF causes the current identity value of the table to be set to the current maximum value in the identity column. In our example, the current identity value was set to 11015. If you now add another row to the table, the order ID will be set to 11016.

INSERT EXEC

With the INSERT EXEC statement, you can insert the result set (or sets) returned by a dynamic batch or a stored procedure into the specified target table. Much like the INSERT VALUES and INSERT SELECT statements, INSERT EXEC supports specifying an optional target column list, and allows omitting columns that accept their values automatically.

To demonstrate the INSERT EXEC statement, the following example uses a procedure called Sales.OrdersForCountry, which accepts a ship country as input and returns orders shipped to the input country. Run the following code to create the Sales.OrdersForCountry procedure:

```
DROP PROC IF EXISTS Sales.OrdersForCountry;
GO

CREATE PROC Sales.OrdersForCountry
  @country AS NVARCHAR(15)
AS

SELECT orderid, custid, empid, orderdate, shipcountry, freight
FROM Sales.Orders
WHERE shipcountry = @country;
GO
```

Run the following code to invoke the stored procedure with Portugal as the input country, and insert the result of the procedure into the Sales.MyOrders table:

```
SET IDENTITY_INSERT Sales.MyOrders ON;

INSERT INTO Sales.MyOrders(orderid, custid, empid, orderdate, shipcountry, freight)
  EXEC Sales.OrdersForCountry
    @country = N'Portugal';

SET IDENTITY_INSERT Sales.MyOrders OFF;
```

Here as well, the code turns on the IDENTITY_INSERT option against the target table so that the INSERT statement can specify the values for the identity column instead of letting the property assign those.

Query the table after running the INSERT statement and notice the new additions of the orders that were shipped to Portugal:

```
SELECT * FROM Sales.MyOrders;
```

INSERT EXEC works even when the source dynamic batch or stored procedure has more than one query. But that's as long as all queries return result sets that are compatible with the target table definition.

SELECT INTO

The SELECT INTO statement involves a query (the SELECT part) and a target table (the INTO part). The statement creates the target table based on the definition of the source and inserts the result rows from the query into that table. The statement copies from the source some aspects of the data definition like the column names, types, nullability, and identity property, in addition to the data itself. Certain aspects of the data definition aren't copied like indexes, constraints, triggers, permissions, and others. If you want to include these aspects, you need to script them from the source and apply them to the target.

The following code shows an example for a SELECT INTO statement that queries the Sales.Orders table returning orders shipped to Norway, creates a target table called Sales.MyOrders, and stores the query's result in the target table:

```
DROP TABLE IF EXISTS Sales.MyOrders;

SELECT orderid, custid, orderdate, shipcountry, freight
INTO Sales.MyOrders
FROM Sales.Orders
WHERE shipcountry = N'Norway';
```

As mentioned, the SELECT INTO statement creates the target table based on the definition of the source. You don't have direct control over the definition of the target. If you want target columns to be defined different than the source, you need to apply some manipulation.

For example, the source orderid column has an identity property, and hence the target column is defined with an identity property as well. If you want the target column not to have the property, you need to apply some kind of manipulation, like orderid + 0 AS orderid. Note that after you apply manipulation, the target column definition allows NULLs. If you want the target column to be defined as not allowing NULLs, you need to use the ISNULL function, returning a non-NULL value in case the source is a NULL. This is just an artificial expression that lets SQL Server know that the outcome cannot be NULL and, hence, the column can be defined as not enabling NULLs. For example, you could use an expression such as this one: ISNULL(orderid + 0, -1) AS orderid.

Similarly, the source custid column is defined in the source as allowing NULLs. To make the target column be defined as NOT NULL, use the expression ISNULL(custid, -1) AS custid.

If you want the target column's type to be different than the source, you can use the CAST or CONVERT functions. But remember that in such a case, the target column definition enables NULLs even if the source column disallowed NULLs, because you applied manipulation to the source column. As with the previous examples, you can use the ISNULL function to make SQL Server define the target column as not enabling NULLs. For example, to convert the orderdate column from its source type DATETIME to DATE in the target, and disallow NULLs, use the expression ISNULL(CAST(orderdate AS DATE), '19000101') AS orderdate.

To put it all together, the following code uses a query similar to the previous example, only defining the orderid column without the identity property as NOT NULL, the custid column as NOT NULL, and the orderdate column as DATE NOT NULL:

```
DROP TABLE IF EXISTS Sales.MyOrders;

SELECT
  ISNULL(orderid + 0, -1) AS orderid, -- get rid of identity property
                                      -- make column NOT NULL
  ISNULL(custid, -1) AS custid, -- make column NOT NULL
  empid,
  ISNULL(CAST(orderdate AS DATE), '19000101') AS orderdate,
  shipcountry, freight
INTO Sales.MyOrders
FROM Sales.Orders
WHERE shipcountry = N'Norway';
```

Remember that SELECT INTO does not copy constraints from the source table, so if you need those, it's your responsibility to define them in the target. For example, the following code defines a primary key constraint in the target table:

```
ALTER TABLE Sales.MyOrders
  ADD CONSTRAINT PK_MyOrders PRIMARY KEY(orderid);
```

Query the table to see the result of the SELECT INTO statement:

```
SELECT * FROM Sales.MyOrders;
```

You get the following output:

```
orderid  custid  empid  orderdate  shipcountry  freight
-------- ------- ------ ---------- ------------ --------
10387    70      1      2014-12-18 Norway       93.63
10520    70      7      2015-04-29 Norway       13.37
10639    70      7      2015-08-20 Norway       38.64
10831    70      3      2016-01-14 Norway       72.19
10909    70      1      2016-02-26 Norway       53.05
11015    70      2      2016-04-10 Norway       4.62
```

One of the benefits of using SELECT INTO is that when the database's recovery model is not set to full, but instead to either simple or bulk logged, the statement uses an optimized logging mode. This can potentially result in a faster insert compared to when full logging is used. You can find details about recovery models at *https://msdn.microsoft.com/en-us/library/ms189275.aspx*. You can find further details on data loading performance at *https://msdn.microsoft.com/en-us/library/dd425070.aspx*.

The SELECT INTO statement also has drawbacks. One of them is that you have only limited control over the definition of the target table. Earlier in this lesson, you reviewed how to control the definition of the target columns indirectly. But some things you simply cannot control—for example the filegroup of the target table.

Also, remember that SELECT INTO involves both creating a table and populating it with data. This means that both the metadata related to the target table and the data are exclusively

locked until the SELECT INTO transaction finishes. As a result, you can run into blocking situations due to conflicts related to both data and metadata access.

When you are done, run the following code for cleanup:

```
DROP TABLE IF EXISTS Sales.MyOrders;
```

Updating data

T-SQL supports the UPDATE statement to enable you to update existing data in your tables. In this section, you review both the standard UPDATE statement and also about a few T-SQL extensions to the standard. You also review modifying data by using joins. You also review nondeterministic updates. Finally, you review how to update with variables, and how all-at-once operations affect updates.

Both the current section, which covers updating data, and the next one, which covers deleting data, use sample data involving tables called Sales.MyCustomers with customer data, Sales.MyOrders with order data, and Sales.MyOrderDetails with order lines data. These tables are made as initial copies of the tables Sales.Customers, Sales.Orders, and Sales.OrderDetails from the TSQLV4 sample database. By working with copies of the original tables, you can safely run code samples that update and delete rows without worrying about making changes to the original tables. Use the following code to create and populate the sample tables:

```
DROP TABLE IF EXISTS Sales.MyOrderDetails, Sales.MyOrders, Sales.MyCustomers;

SELECT * INTO Sales.MyCustomers FROM Sales.Customers;
ALTER TABLE Sales.MyCustomers
  ADD CONSTRAINT PK_MyCustomers PRIMARY KEY(custid);

SELECT * INTO Sales.MyOrders FROM Sales.Orders;
ALTER TABLE Sales.MyOrders
  ADD CONSTRAINT PK_MyOrders PRIMARY KEY(orderid);

SELECT * INTO Sales.MyOrderDetails FROM Sales.OrderDetails;
ALTER TABLE Sales.MyOrderDetails
  ADD CONSTRAINT PK_MyOrderDetails PRIMARY KEY(orderid, productid);
```

UPDATE statement

T-SQL supports the standard UPDATE statement, which enables you to update existing rows in a table. The standard UPDATE statement has the following form:

```
UPDATE <target table>
  SET <col 1> = <expression 1>,
      ...,
      <col n> = <expression n>
WHERE <predicate>;
```

You specify the target table name in the UPDATE clause. If you want to filter a subset of rows, you indicate a WHERE clause with a predicate. Only rows for which the predicate evaluates to true are updated. Rows for which the predicate evaluates to false or unknown are not

affected. An UPDATE statement without a WHERE clause affects all rows. You assign values to target columns in the SET clause. The source expressions can involve columns from the table, in which case their values before the update are used.

As an example, you modify rows in the Sales.MyOrderDetails table representing order lines associated with order 10251. First, query those rows to examine their state prior to the update:

```
SELECT *
FROM Sales.MyOrderDetails
WHERE orderid = 10251;
```

You get the following output:

```
orderid     productid   unitprice             qty    discount
----------- ----------- --------------------- ------ ---------
10251       22          16.80                 6      0.050
10251       57          15.60                 15     0.050
10251       65          16.80                 20     0.000
```

The following code demonstrates an UPDATE statement that adds a five percent discount to these order lines:

```
UPDATE Sales.MyOrderDetails
  SET discount += 0.05
WHERE orderid = 10251;
```

Notice the use of the compound assignment operator discount += 0.05. This assignment is equivalent to discount = discount + 0.05. T-SQL supports such enhanced operators for all binary assignment operators: += (add), -= (subtract), *= (multiply), /= (divide), %= (modulo), &= (bitwise and), |= (bitwise or), ^= (bitwise xor), += (concatenate).

Query again the order lines associated with order 10251 to see their state after the update:

```
SELECT *
FROM Sales.MyOrderDetails
WHERE orderid = 10251;
```

You get the following output showing an increase of five percent in the discount:

```
orderid     productid   unitprice             qty    discount
----------- ----------- --------------------- ------ ---------
10251       22          16.80                 6      0.100
10251       57          15.60                 15     0.100
10251       65          16.80                 20     0.050
```

Use the following code to reduce the discount in the aforementioned order lines by five percent:

```
UPDATE Sales.MyOrderDetails
  SET discount -= 0.05
WHERE orderid = 10251;
```

These rows should now be back to their original state before the first update.

EXAM TIP

If you're using a cursor to iterate through rows of a table (you can find details on cursors at *https://msdn.microsoft.com/en-us/library/ms180169.aspx*), you can modify the table row that the cursor is currently positioned on by using the filter WHERE CURRENT OF <cursor_name>. For example, suppose that you iterate through rows of a table called MyTable using a cursor called MyCursor. Based on some condition that is met, you want to increase the current row's discount by five percent. You achieve this using the statement:

```
UPDATE dbo.MyTable SET discount += 0.05 WHERE CURRENT OF MyCursor;
```

UPDATE based on join

Standard SQL doesn't support using joins in UPDATE statements, but T-SQL does. The idea is that you might want to update rows in a table, and refer to related rows in other tables for filtering and assignment purposes.

As an example, suppose that you want to add a five percent discount to order lines associated with orders placed by customers from Norway. The rows you need to modify are in the Sales.MyOrderDetails table. But the information you need to examine for filtering purposes is in rows in the Sales.MyCustomers table. In order to match a customer with its related order lines, you need to join Sales.MyCustomers with Sales.MyOrders, and then join the result with Sales.MyOrderDetails. Note that it's not sufficient to examine the shipcountry column in Sales.MyOrders; instead, you must check the country column in Sales.MyCustomers.

Based on your knowledge of joins, if you wanted to write a SELECT statement returning the order lines that are the target for the update, you would write a query like the following one:

```
SELECT OD.*
FROM Sales.MyCustomers AS C
  INNER JOIN Sales.MyOrders AS O
    ON C.custid = O.custid
  INNER JOIN Sales.MyOrderDetails AS OD
    ON O.orderid = OD.orderid
WHERE C.country = N'Norway';
```

This query identifies 16 order lines, all currently with a discount value of 0.000.

In order to perform the desired update, simply replace the SELECT clause from the last query with an UPDATE clause, indicating the alias of the table that is the target for the UPDATE (OD in this case), and the assignment in the SET clause, as follows:

```
UPDATE OD
  SET OD.discount += 0.05
FROM Sales.MyCustomers AS C
  INNER JOIN Sales.MyOrders AS O
    ON C.custid = O.custid
  INNER JOIN Sales.MyOrderDetails AS OD
    ON O.orderid = OD.orderid
WHERE C.country = N'Norway';
```

Note that you can refer to elements from all tables involved in the statement in the source expressions, but you're allowed to modify only one target table at a time. Rerun the SELECT query to examine the affected order lines, and you find that they now have a discount value of 0.050.

To get the previous order lines back to their original state, run an UPDATE statement that reduces the discount by five percent:

```
UPDATE OD
  SET OD.discount -= 0.05
FROM Sales.MyCustomers AS C
  INNER JOIN Sales.MyOrders AS O
    ON C.custid = O.custid
  INNER JOIN Sales.MyOrderDetails AS OD
    ON O.orderid = OD.orderid
WHERE C.country = N'Norway';
```

Nondeterministic UPDATE

You should be aware that the proprietary T-SQL UPDATE syntax based on joins could be nondeterministic. The statement is nondeterministic when multiple source rows match one target row. Unfortunately, in such a case, SQL Server doesn't generate an error or even a warning. Instead, SQL Server silently performs a nondeterministic UPDATE where it arbitrarily chooses one of the source rows.

As an example, the following query matches customers with their related orders, returning the customers' postal codes, as well as shipping postal codes from related orders:

```
SELECT C.custid, C.postalcode, O.shippostalcode
FROM Sales.MyCustomers AS C
  INNER JOIN Sales.MyOrders AS O
    ON C.custid = O.custid
ORDER BY C.custid;
```

This query generates the following output:

```
custid      postalcode shippostalcode
----------- ---------- --------------
1           10092      10154
1           10092      10156
1           10092      10155
1           10092      10154
1           10092      10154
1           10092      10154
2           10077      10182
2           10077      10181
...
```

Each customer row is repeated in the output for each matching order. This means that each customer's only postal code is repeated in the output as many times as the number of matching orders. It's important for the purposes of this example to remember that there is only one postal code per customer. The shipping postal code is associated with an order, so as

you can realize, there can be multiple distinct shipping postal codes per customer. With this in mind, consider the following UPDATE statement:

```
UPDATE C
  SET C.postalcode = O.shippostalcode
FROM Sales.MyCustomers AS C
  INNER JOIN Sales.MyOrders AS O
    ON C.custid = O.custid;
```

There are 89 customers that have matching orders—some with multiple matches. SQL Server doesn't generate an error though; instead it arbitrarily chooses for each target row which source row is to be considered for the update, returning the following message:

```
(89 row(s) affected)
```

Query the rows from the Sales.Customers table after the update:

```
SELECT custid, postalcode
FROM Sales.MyCustomers
ORDER BY custid;
```

This generated the following output on one system, but your results could be different:

```
custid      postalcode
----------- ----------
1           10154
2           10182
...

(91 row(s) affected)
```

Note that the table has 91 rows, but because only 89 of those customers have related orders, the previous UPDATE statement affected 89 rows.

As to which source row gets chosen for each target row, the choice isn't exactly random, but arbitrary; in other words, it's optimization-dependent. At any rate, you do not have any logical elements in the language to control this choice. The recommended approach is simply not to use such nondeterministic UPDATE statements, rather have logic in your solution to break ties.

For example, suppose that you want to update the customer's postal code with the shipping postal code from the customer's first order (based on the sort order of orderdate, orderid). You can achieve this using the following code:

```
UPDATE C
  SET C.postalcode = A.shippostalcode
FROM Sales.MyCustomers AS C
  CROSS APPLY (SELECT TOP (1) O.shippostalcode
               FROM Sales.MyOrders AS O
               WHERE O.custid = C.custid
               ORDER BY orderdate, orderid) AS A;
```

The book covers the APPLY operator and correlated subqueries in Chapter 2. For now, suffice to say that for each customer, the operator applies a subquery that identifies its most recent order. Customers who don't have orders aren't affected. Can't wait till Chapter 2 and feel like watching a movie? Grab some popcorn and watch the following video seminar about the APPLY operator at *http://aka.ms/BoostTSQL*.

SQL Server generates the following message:

```
(89 row(s) affected)
```

Query the Sales.MyCustomers table after the update:

```
SELECT custid, postalcode
FROM Sales.MyCustomers
ORDER BY custid;
```

You get the following output:

```
custid      postalcode
----------- ----------
1           10154
2           10180
...

(91 row(s) affected)
```

If you want to use the most-recent order as the source for the update, simply use descending sort order in both columns: ORDER BY orderdate DESC, orderid DESC.

UPDATE with a variable

Sometimes you need to modify a row and also collect the result of the modified columns into variables. You can handle such a need with a combination of UPDATE and SELECT statements, but this would require two visits to the row. T-SQL supports a specialized UPDATE syntax that allows achieving the task by using one statement and one visit to the row.

As an example, run the following query to examine the current state of the order line associated with order 10250 and product 51:

```
SELECT *
FROM Sales.MyOrderDetails
WHERE orderid = 10250
  AND productid = 51;
```

This code generates the following output:

```
orderid     productid   unitprice             qty    discount
----------- ----------- --------------------- ------ ---------
10250       51          42.40                 35     0.150
```

Suppose that you need to modify the row, increasing the discount by five percent, and collect the new discount into a variable called @newdiscount. You can achieve this using a single UPDATE statement, as follows.

```
DECLARE @newdiscount AS NUMERIC(4, 3) = NULL;

UPDATE Sales.MyOrderDetails
  SET @newdiscount = discount += 0.05
WHERE orderid = 10250
  AND productid = 51;

SELECT @newdiscount;
```

As you can see, the UPDATE and WHERE clauses are similar to those you use in normal UPDATE statements. But the SET clause uses the assignment @newdiscount = discount += 0.05, which is equivalent to using @newdiscount = discount = discount + 0.05. The statement assigns the result of discount + 0.05 to discount, and then assigns the result to the variable @newdiscount. The last SELECT statement in the code returns the new discount 0.200.

When you're done, issue the following code to undo the last change:

```
UPDATE Sales.MyOrderDetails
  SET discount -= 0.05
WHERE orderid = 10250
  AND productid = 51;
```

UPDATE all-at-once

Earlier in the book as part of the discussion about logical query processing I explained that expressions that appear in the same logical phase are treated as a set, in an *all-at-once* manner. The all-at-once concept also has implications on UPDATE statements. To demonstrate those implications, this section uses a table called T1. Use the following code to create the table T1 and insert a row into it:

```
DROP TABLE IF EXISTS dbo.T1;

CREATE TABLE dbo.T1
(
  keycol INT NOT NULL
    CONSTRAINT PK_T1 PRIMARY KEY,
  col1 INT NOT NULL,
  col2 INT NOT NULL
);

INSERT INTO dbo.T1(keycol, col1, col2) VALUES(1, 100, 0);
```

Next, examine the following code but don't run it yet:

```
DECLARE @add AS INT = 10;

UPDATE dbo.T1
  SET col1 += @add, col2 = col1
WHERE keycol = 1;

SELECT * FROM dbo.T1;
```

Can you guess what should be the value of col2 in the modified row after the update? If you guessed 110, you were not thinking of the assignments as a set, all-at-once. All assignments use the original values of the row as the source values, irrespective of their order of appearance. So the assignment col2 = col1 doesn't get the col1 value after the change, but rather before the change—namely 100. To verify this, run the previous code.

You get the following output:

```
keycol      col1        col2
----------- ----------- -----------
1           110         100
```

When you're done, run the following code for cleanup:

```
DROP TABLE IF EXISTS dbo.T1;
```

Deleting data

T-SQL supports two statements that you can use to delete rows from a table: DELETE and TRUNCATE TABLE. This section describes these statements, the differences between them, and different aspects of working with them.

This section uses the same sample data that was used in the previous section. As a reminder, the sample data involves the tables Sales.MyCustomers, Sales.MyOrders, and Sales.MyOrderDetails, which are initially created as copies of the tables Sales.Customers, Sales.Orders, and Sales.OrderDetails, respectively. Use the following code to recreate tables and repopulate them with sample data:

```
DROP TABLE IF EXISTS Sales.MyOrderDetails, Sales.MyOrders, Sales.MyCustomers;

SELECT * INTO Sales.MyCustomers FROM Sales.Customers;
ALTER TABLE Sales.MyCustomers
  ADD CONSTRAINT PK_MyCustomers PRIMARY KEY(custid);

SELECT * INTO Sales.MyOrders FROM Sales.Orders;
ALTER TABLE Sales.MyOrders
  ADD CONSTRAINT PK_MyOrders PRIMARY KEY(orderid);

SELECT * INTO Sales.MyOrderDetails FROM Sales.OrderDetails;
ALTER TABLE Sales.MyOrderDetails
  ADD CONSTRAINT PK_MyOrderDetails PRIMARY KEY(orderid, productid);
```

DELETE statement

With the DELETE statement, you can delete rows from a table. You can optionally specify a predicate to restrict the rows to be deleted. The general form of a DELETE statement looks like the following:

```
DELETE FROM <table>
WHERE <predicate>;
```

If you don't specify a predicate, all rows from the target table are deleted. As with unqualified updates, you need to be especially careful about accidentally deleting all rows by highlighting only the DELETE part of the statement, missing the WHERE part.

The following example deletes all order lines containing product ID 11 from the Sales.MyOrderDetails table:

```
DELETE FROM Sales.MyOrderDetails
WHERE productid = 11;
```

You get a message indicating that 38 rows were affected.

The tables used by the examples in this chapter are very small, but in a more realistic production environment, the volumes of data are likely to be much bigger. A DELETE statement is fully logged (you can find details on the transaction log at *https://msdn.microsoft.com/en-us/library/ms190925.aspx*) and as a result, large deletes can take a long time to complete, and much longer to roll back if you need to terminate them. Such large deletes can cause the transaction log to increase in size dramatically during the process. They can also result in lock escalation, meaning that SQL Server escalates fine-grained locks like row or page locks to a full-blown table lock. Such escalation can result in blocking access to all table data by other processes. You can find details on locking at *https://technet.microsoft.com/en-us/library/ms190615(v=sql.105).aspx*.

To prevent the aforementioned problems from happening, you can split your large delete into smaller chunks. You can achieve this by using a DELETE statement with a TOP option that limits the number of affected rows in a loop. Here's an example for implementing such a solution:

```
WHILE 1 = 1
BEGIN
  DELETE TOP (1000) FROM Sales.MyOrderDetails
  WHERE productid = 12;

  IF @@rowcount < 1000 BREAK;
END
```

As you can see, the code uses an infinite loop (WHILE 1 = 1 is always true). In each iteration, a DELETE statement with a TOP option limits the number of affected rows to no more than 1,000 at a time. Then the IF statement checks if the number of affected rows is less than 1,000; in such a case, the last iteration deleted the last chunk of qualifying rows. After the last chunk of rows has been deleted, the code breaks from the loop. With this sample data, there are only 14 qualifying rows in total. So if you run this code, it is done after one round, break from the loop, and return. But with a large number of qualifying rows, say, millions, you'd very likely be better off with such a solution.

EXAM TIP

Similar to the UPDATE WHERE CURRENT OF syntax, if you're using a cursor to iterate through rows of a table, you can delete the table row that the cursor is currently positioned on by using the syntax DELETE WHERE CURRENT OF. For example, suppose that you iterate through rows of a table called MyTable using a cursor called MyCursor. Based on some condition that is met, you want to delete the current row. You achieve this using the statement:

```
DELETE FROM dbo.MyTable WHERE CURRENT OF MyCursor;
```

TRUNCATE TABLE statement

TRUNCATE TABLE is an optimized statement that deletes all rows from the target table or partition. Unlike the DELETE statement, the TRUNCATE TABLE statement doesn't support a filter. Also, whereas the DELETE statement is fully logged and therefore tends to be quite slow, the TRUNCATE table statement uses an optimized logging mode and therefore is significantly faster.

For example, the following statement truncates the table Sales.MyOrderDetails:

```
TRUNCATE TABLE Sales.MyOrderDetails;
```

Suppose that you had a partitioned table called MyTable and you wanted to truncate partitions 1, 2 and 11 to 20. You would achieve this with the following code:

```
TRUNCATE TABLE MyTable WITH ( PARTITIONS(1, 2, 11 TO 20) );
```

Besides the performance difference and the fact that TRUNCATE TABLE doesn't support a filter, there are a few additional differences compared to the DELETE statement:

- You cannot assign direct TRUNCATE TABLE permissions, rather at minimum you need ALTER permission on the target table. A common workaround is to place the TRUNCATE TABLE statement in a module, like a stored procedure, and assign the required permission to the module using the EXECUTE AS clause.
- If there's a column with an identity property in the target table, DELETE doesn't reset the property whereas TRUNCATE TABLE does.
- If there are any foreign keys pointing to the target table, a DELETE statement is supported as long as there are no related rows in the referencing table, but a TRUNCATE TABLE statement isn't. You need to first drop the foreign keys, truncate the table, and then recreate the foreign keys.
- If there are any indexed views based on the table, a DELETE statement is supported whereas a TRUNCATE TABLE statement isn't.

Clearly, if you need to delete all rows from a table or a partition but leave the table definition in place, the recommended tool to use is the TRUNCATE TABLE statement.

DELETE based on a join

Much like the proprietary syntax that T-SQL supports for an UPDATE statement based on a join, T-SQL supports similar syntax for a DELETE statement based on a join. The idea is to allow you to delete rows from one table based on the presence of related rows in other tables, with the ability to apply a filter predicate that is based on attributes in the related tables.

As an example, the following statement deletes orders placed by customers from the US:

```
DELETE FROM O
FROM Sales.MyOrders AS O
  INNER JOIN Sales.MyCustomers AS C
    ON O.custid = C.custid
WHERE C.country = N'USA';
```

Notice that there are two FROM clauses. The second is mandatory and is similar to the FROM clause in a SELECT statement. That's where you apply table operators like joins. The first FROM clause appears right after the DELETE clause and is optional. That's where you specify the target for the delete. In our case it's the alias O representing the Sales.MyOrders table.

When you're done, run the following code for cleanup:

```
DROP TABLE IF EXISTS Sales.MyOrderDetails, Sales.MyOrders, Sales.MyCustomers;
```

Merging data

With the MERGE statement, you can merge data from a source table into a target table. The statement has many practical uses in both online transaction processing (OLTP) scenarios and in data warehousing ones. As an example of an OLTP use case, suppose that you have a table that isn't updated directly by your application; instead, you get the delta of changes periodically from an external system. You first load the delta of changes into a staging table, and then use the staging table as the source for the merge operation into the target.

As an example for a data warehousing scenario, suppose that you maintain aggregated views of the data in your data warehouse. Using the MERGE statement, you can apply changes that were applied to detail rows into the aggregated form.

These are just a couple of typical use cases; there are many more. This lesson describes the MERGE statement and its different options, and demonstrates its use through examples.

Using the MERGE statement

With the MERGE statement, you can merge data from a source table or table expression into a target table. This statement is mostly standard, with one proprietary extension by Microsoft of a clause called WHEN NOT MATCHED BY SOURCE. The general form of the MERGE statement is as follows:

```
MERGE INTO <target table> AS TGT
USING <SOURCE TABLE> AS SRC
  ON <merge predicate>
WHEN MATCHED [AND <predicate>] -- two clauses allowed:
  THEN <action> -- one with UPDATE one with DELETE
WHEN NOT MATCHED [BY TARGET] [AND <predicate>] -- one clause allowed:
  THEN INSERT... -- if indicated, action must be INSERT
WHEN NOT MATCHED BY SOURCE [AND <predicate>] -- two clauses allowed:
  THEN <action>; -- one with UPDATE one with DELETE
```

The following are the clauses of the statement and their roles:

- **MERGE INTO <target table>** This clause defines the target table for the operation. You can alias the table in this clause if you want.
- **USING <source table>** This clause defines the source table for the operation. You can alias the table in this clause if you want. Note that the USING clause is designed similar to a FROM clause in a SELECT query, meaning that in this clause you can define table operators like joins, refer to a table expression like a derived table or a common table expression (CTE), or even refer to a table function like OPENROWSET. The outcome of the USING clause is eventually a table result, and that table is considered the source of the merge operation.
- **ON <merge predicate>** In this clause, you specify a predicate that matches rows between the source and the target and defines whether a source row is or isn't matched by a target row. Note that this clause isn't a filter like the ON clause in a join.
- **WHEN MATCHED [AND <predicate>] THEN <action>** This clause defines an action to take when a source row is matched by a target row. Because a target row exists, an INSERT action isn't allowed in this clause. The two actions that are enabled are UPDATE and DELETE. If you want to apply different actions in different conditions, you can specify two WHEN MATCHED clauses, each with a different additional predicate to determine when to apply an UPDATE and when to apply a DELETE.
- **WHEN NOT MATCHED [BY TARGET] [AND <predicate>] THEN <action>** This clause defines what action to take when a source row is not matched by a target row. Because a target row does not exist, the only action allowed in this clause (if you choose to include this clause in the statement) is INSERT. Using UPDATE or DELETE holds no meaning when a target row doesn't exist. You can still add an additional predicate that must be true in order to perform the action.
- **WHEN NOT MATCHED BY SOURCE [AND <predicate>] THEN <action>** This clause is a proprietary extension by Microsoft to the standard MERGE statement syntax. It defines an action to take when a target row exists, but it is not matched by a source row. Because a target row exists, you can apply either an UPDATE or a DELETE, but not an INSERT. If you want, you can have two such clauses with different additional predicates that define when to use an UPDATE and when to use a DELETE.

To demonstrate examples of the MERGE statement, this section uses the Sales.MyOrders table and the Sales.SeqOrderIDs sequence. Use the following code to create these objects.

```
DROP TABLE IF EXISTS Sales.MyOrders;
DROP SEQUENCE IF EXISTS Sales.SeqOrderIDs;

CREATE SEQUENCE Sales.SeqOrderIDs AS INT
  MINVALUE 1
  CACHE 10000;

CREATE TABLE Sales.MyOrders
(
  orderid INT NOT NULL
    CONSTRAINT PK_MyOrders_orderid PRIMARY KEY
    CONSTRAINT DFT_MyOrders_orderid
      DEFAULT(NEXT VALUE FOR Sales.SeqOrderIDs),
  custid  INT NOT NULL
    CONSTRAINT CHK_MyOrders_custid CHECK(custid > 0),
  empid   INT NOT NULL
    CONSTRAINT CHK_MyOrders_empid CHECK(empid > 0),
  orderdate DATE NOT NULL
);
```

Notice that the sequence is defined to start with the value 1, and uses a cache size of 10,000 for performance reasons. The cache size defines how frequently to write a recoverable value to disk. To request a new key from the sequence, you use the function NEXT VALUE FOR <sequence_name>. Our code defines a default constraint with the function call for the orderid column to automate the creation of keys when new rows are inserted.

Suppose that you need to define a stored procedure that accepts as input parameters attributes of an order. If an order with the input order ID already exists in the Sales.MyOrders table, you need to update the row, setting the values of the nonkey columns to the new ones. If the order ID doesn't exist in the target table, you need to insert a new row. Because this book doesn't cover stored procedures until Chapter 3, the examples in this section use local variables for now. A MERGE statement in a stored procedure simply refers to the procedure's input parameters instead of the local variables.

The first things to identify in a MERGE statement are the target and the source tables. The target is easy—it's the Sales.MyOrders table. The source is supposed to be a table or table expression, but in this case, it's just a set of input parameters making an order. To turn the inputs into a table expression, you can define a derived table based on the VALUES clause, which is also known as a table value constructor. The following MERGE statement updates the target row if the source key exists in the target, and inserts a new row if it doesn't:

```
DECLARE
  @orderid   AS INT  = 1, @custid    AS INT  = 1,
  @empid     AS INT  = 2, @orderdate AS DATE = '20170212';

MERGE INTO Sales.MyOrders WITH (SERIALIZABLE) AS TGT
USING (VALUES(@orderid, @custid, @empid, @orderdate))
       AS SRC( orderid,  custid,  empid,  orderdate)
  ON SRC.orderid = TGT.orderid
WHEN MATCHED THEN
  UPDATE
    SET TGT.custid    = SRC.custid,
```

```
        TGT.empid     = SRC.empid,
        TGT.orderdate = SRC.orderdate
WHEN NOT MATCHED THEN
  INSERT VALUES(SRC.orderid, SRC.custid, SRC.empid, SRC.orderdate);
```

Observe that the MERGE predicate compares the source order ID with the target order ID. When a match is found (the source order ID is matched by a target order ID), the MERGE statement performs an UPDATE action that updates the values of the nonkey columns in the target to those from the respective source row.

When a match isn't found (the source order ID is not matched by a target order ID), the MERGE statement inserts a new row with the source order information into the target.

> ***IMPORTANT*** **MERGE CONFLICTS**
>
> **Suppose that a certain key K doesn't yet exist in the target table. Two processes, P1 and P2, run a MERGE statement such as the previous one at the same time with the same source key K. It is normally possible for the MERGE statement issued by P1 to insert a new row with the key K between the points in time when the MERGE statement issued by P2 checks whether the target already has that key and inserts rows. In such a case, the MERGE statement issued by P2 fails due to a primary key violation. To prevent such a failure, use the hint SERIALIZABLE or HOLDLOCK (both have equivalent meanings) against the target as shown in the previous statement. This hint means that the statement uses a serializable isolation level to serialize access to the data, meaning that once you get access to the data, it's as if you're the only one interacting with it.**

Remember that you cleared the Sales.MyOrders table at the beginning of this section. So if you run the previous code for the first time, it performs an INSERT action against the target. If you run it a second time, it performs an UPDATE action.

Regarding the second run of the code, notice that it's a waste to issue an UPDATE action when the source and target rows are completely identical. An update costs you resources and time, and furthermore, if there are any triggers or auditing activity taking place, they consider the target row as updated. There is a way to avoid such an update when there's no real value change. Remember that each WHEN clause in the MERGE statement allows an additional predicate that must be true in order for the respective action to be applied. You can add a predicate that says that at least one of the nonkey column values in the source and the target must be different in order to apply the UPDATE action. Your code would look like the following:

```
DECLARE
  @orderid   AS INT  = 1, @custid    AS INT  = 1,
  @empid     AS INT  = 2, @orderdate AS DATE = '20170212';

MERGE INTO Sales.MyOrders WITH (SERIALIZABLE) AS TGT
USING (VALUES(@orderid, @custid, @empid, @orderdate))
       AS SRC( orderid,  custid,  empid,  orderdate)
  ON SRC.orderid = TGT.orderid
```

```
WHEN MATCHED AND (   TGT.custid    <> SRC.custid
                  OR TGT.empid     <> SRC.empid
                  OR TGT.orderdate <> SRC.orderdate) THEN
  UPDATE
    SET TGT.custid    = SRC.custid,
        TGT.empid     = SRC.empid,
        TGT.orderdate = SRC.orderdate
WHEN NOT MATCHED THEN
  INSERT VALUES(SRC.orderid, SRC.custid, SRC.empid, SRC.orderdate);
```

Note that if any of the nonkey columns use NULLs, you need to add extra logic for correct NULL treatment. For instance, suppose that the custid column used NULLs. The predicates for this column would be:

```
    TGT.custid <> SRC.custid
  OR (TGT.custid IS NULL AND SRC.custid IS NOT NULL)
  OR (TGT.custid IS NOT NULL AND SRC.custid IS NULL)
```

Alternatively, similar to the way you matched rows in a join using a set operator, you can identify a difference here between the source and target rows as follows:

```
WHEN MATCHED AND EXISTS( SELECT SRC.* EXCEPT SELECT TGT.* ) THEN UPDATE
```

Remember that a set operator uses distinctness in the comparison, and one NULL is distinct from a non-NULL value, but not distinct from another NULL. When there is a difference between the source and target rows, the EXCEPT operator returns one row, the EXISTS predicate returns true, and the MERGE statement applies the update. When the source and target rows are the same, the set operator yields an empty set, EXISTS returns false, and the MERGE statement doesn't proceed with the update.

What's interesting about the USING clause where you define the source for the MERGE operation is that it's designed like the FROM clause in a SELECT statement. This means that you can define table operators like JOIN, APPLY, PIVOT, and UNPIVOT; and use table expressions like derived tables, CTEs, views, inline table functions, and even table functions like OPENROWSET and OPENXML. You can refer to real tables, temporary tables, or table variables as the source. Ultimately, the USING clause returns a table result, and that table result is used as the source for the MERGE statement.

T-SQL extends standard SQL by supporting a third clause called WHEN NOT MATCHED BY SOURCE. With this clause, you can define an action to take against the target row when the target row exists but is not matched by a source row. The allowed actions are UPDATE and DELETE. For example, suppose that you want to add such a clause to the last example to indicate that if a target row exists and it is not matched by a source row, you want to delete the target row. Here's how your MERGE statement would look (this time using a table variable with multiple orders as the source):

```
DECLARE @Orders AS TABLE
(
  orderid   INT  NOT NULL PRIMARY KEY,
  custid    INT  NOT NULL,
  empid     INT  NOT NULL,
```

```
  orderdate DATE NOT NULL
);

INSERT INTO @Orders(orderid, custid, empid, orderdate)
  VALUES (2, 1, 3, '20170212'),
         (3, 2, 2, '20170212'),
         (4, 3, 5, '20170212');

-- update where exists (only if different), insert where not exists,
-- delete when exists in target but not in source
MERGE INTO Sales.MyOrders AS TGT
USING @Orders AS SRC
  ON SRC.orderid = TGT.orderid
WHEN MATCHED AND EXISTS( SELECT SRC.* EXCEPT SELECT TGT.* ) THEN
  UPDATE
    SET TGT.custid    = SRC.custid,
        TGT.empid     = SRC.empid,
        TGT.orderdate = SRC.orderdate
WHEN NOT MATCHED THEN
  INSERT VALUES(SRC.orderid, SRC.custid, SRC.empid, SRC.orderdate)
WHEN NOT MATCHED BY SOURCE THEN
  DELETE;
```

Before you ran this statement, only one row in the table had order ID 1. So the statement inserted the three rows with order IDs 2, 3, and 4, and deleted the row that had order ID 1. Query the current state of the table:

```
SELECT * FROM Sales.MyOrders;
```

You get the following output with the three remaining rows:

```
orderid     custid      empid       orderdate
----------- ----------- ----------- ----------
2           1           3           2017-02-12
3           2           2           2017-02-12
4           3           5           2017-02-12
```

You can find more information about the MERGE statement at *http://sqlmag.com/sql-server/merge-statement-tips*.

Using the OUTPUT option

T-SQL supports an OUTPUT clause for modification statements, which you can use to return information from modified rows. You can use the output for purposes like auditing, archiving and others. This section covers the OUTPUT clause with the different types of modification statements and demonstrates using the clause through examples. I use the same Sales.MyOrders table and Sales.SeqOrderIDs sequence from the Merging data section in my examples, so make sure you still have them around. Run the following code to clear the table and reset the sequence start value to 1:

```
TRUNCATE TABLE Sales.MyOrders;
ALTER SEQUENCE Sales.SeqOrderIDs RESTART WITH 1;
```

The design of the OUTPUT clause is very similar to that of the SELECT clause in the sense that you can specify expressions and assign them with result column aliases. One difference from the SELECT clause is that, in the OUTPUT clause, when you refer to columns from the modified rows, you need to prefix the column names with the keywords *inserted* or *deleted*. Use the prefix inserted when the rows are inserted rows and the prefix deleted when they are deleted rows. In an UPDATE statement, inserted represents the state of the rows after the update and deleted represents the state before the update.

You can have the OUTPUT clause return a result set back to the caller much like a SELECT does. Or you can add an INTO clause to direct the output rows into a target table. In fact, you can have two OUTPUT clauses if you like—the first with INTO directing the rows into a table, and the second without INTO, returning a result set from the query. If you do use the INTO clause, the target table cannot participate in either side of a foreign key relationship and cannot have triggers defined on it.

INSERT with OUTPUT

The OUTPUT clause can be used in an INSERT statement to return information from the inserted rows. An example for a practical use case is when you have a multi-row INSERT statement that generates new keys by using the identity property or a sequence, and you need to know which new keys were generated.

For example, suppose that you need to query the Sales.Orders table and insert orders shipped to Norway to the Sales.MyOrders table. You are not going to use the original order IDs in the target rows; instead, let the sequence object generate those for you. But you need to get back information from the INSERT statement about which order IDs were generated, plus additional columns from the inserted rows. To achieve this, simply add an OUTPUT clause to the INSERT statement right before the query. List the columns that you need to return from the inserted rows and prefix them with the keyword inserted, as follows:

```
INSERT INTO Sales.MyOrders(custid, empid, orderdate)
  OUTPUT
    inserted.orderid, inserted.custid, inserted.empid, inserted.orderdate
  SELECT custid, empid, orderdate
  FROM Sales.Orders
  WHERE shipcountry = N'Norway';
```

This code generates the following output:

```
orderid     custid      empid       orderdate
----------- ----------- ----------- ----------
1           70          1           2014-12-18
2           70          7           2015-04-29
3           70          7           2015-08-20
4           70          3           2016-01-14
5           70          1           2016-02-26
6           70          2           2016-04-10
```

You can see that the sequence object generated the order IDs 1 through 6 for the new rows. If you need to store the result in a table instead of returning it back to the caller, add an INTO clause with an existing target table name as follows:

```
  OUTPUT
    inserted.orderid, inserted.custid, inserted.empid, inserted.orderdate
    INTO SomeTable(orderid, custid, empid, orderdate)
```

In an INSERT statement you're not allowed to use the deleted prefix given how there are no deleted rows.

DELETE with OUTPUT

You can use the OUTPUT clause to return information from deleted rows in a DELETE statement. You need to prefix the columns that you refer to with the keyword deleted. In a DELETE statement you're not allowed to use the inserted prefix given that there are no inserted rows.

The following example deletes the rows from the Sales.MyOrders table where the employee ID is equal to 1. Using the OUTPUT clause, the code returns the order IDs of the deleted orders:

```
DELETE FROM Sales.MyOrders
  OUTPUT deleted.orderid
WHERE empid = 1;
```

This code generates the following output:

```
orderid
-----------
1
5
```

Remember that if you need to persist the output rows in a table—for example, for archiving purposes—you can add an INTO clause with the target table name.

UPDATE with OUTPUT

You can use the OUTPUT clause to return information from modified rows in an UPDATE statement. With updated rows, you have access to both the old and the new images of the modified rows. To refer to columns from the original state of the row before the update, prefix the column names with the keyword deleted. To refer to columns from the new state of the row after the update, prefix the column names with the keyword inserted.

As an example, the following UPDATE statement adds a day to the order date of all orders that were handled by employee 7:

```
UPDATE Sales.MyOrders
  SET orderdate = DATEADD(day, 1, orderdate)
  OUTPUT
    inserted.orderid,
    deleted.orderdate AS old_orderdate,
    inserted.orderdate AS neworderdate
WHERE empid = 7;
```

The code uses the OUTPUT clause to return the order IDs of the modified rows, in addition to the order dates—both before and after the update. This code generates the following output:

```
orderid     old_orderdate neworderdate
----------- ------------- ------------
2           2015-04-29    2015-04-30
3           2015-08-20    2015-08-21
```

MERGE with OUTPUT

You can use the OUTPUT clause with the MERGE statement, but there are special considerations with this statement. Remember that one MERGE statement can apply different actions against the target table. And suppose that when returning output rows, you need to know which action (INSERT, UPDATE, or DELETE) affected the output row. For this purpose, SQL Server provides you with the $action function. This function returns a string ('INSERT', 'UPDATE', or 'DELETE') indicating the action.

As explained before, you can refer to columns from the deleted rows with the deleted prefix and to columns from the inserted rows with the inserted prefix. Rows affected by an INSERT action have values in the inserted row and NULLs in the deleted row. Rows affected by a DELETE action have NULLs in the inserted row and values in the deleted row. Rows affected by an UPDATE action have values in both. So, for example, if you want to return the key of the affected row (assuming the key itself wasn't modified), you can use the expression COALESCE(inserted.orderid, deleted.orderid).

The following example demonstrates the use of the MERGE statement with the OUTPUT clause, returning the output of the $action function to indicate which action affected the row, and the key of the modified row:

```
MERGE INTO Sales.MyOrders AS TGT
USING (VALUES(1, 70, 1, '20151218'), (2, 70, 7, '20160429'), (3, 70, 7, '20160820'),
              (4, 70, 3, '20170114'), (5, 70, 1, '20170226'), (6, 70, 2, '20170410'))
       AS SRC(orderid, custid, empid, orderdate)
  ON SRC.orderid = TGT.orderid
WHEN MATCHED AND EXISTS( SELECT SRC.* EXCEPT SELECT TGT.* ) THEN
  UPDATE SET TGT.custid    = SRC.custid,
             TGT.empid     = SRC.empid,
             TGT.orderdate = SRC.orderdate
WHEN NOT MATCHED THEN
  INSERT VALUES(SRC.orderid, SRC.custid, SRC.empid, SRC.orderdate)
WHEN NOT MATCHED BY SOURCE THEN
  DELETE
OUTPUT
  $action AS the_action,
  COALESCE(inserted.orderid, deleted.orderid) AS orderid;
```

This code generates the following output:

```
the_action orderid
---------- -----------
INSERT     1
```

```
UPDATE      2
UPDATE      3
UPDATE      4
INSERT      5
UPDATE      6
```

The output shows that two rows were inserted and two were updated.

EXAM TIP

In INSERT, UPDATE, and DELETE statements, you can only refer to columns from the target table in the OUTPUT clause. In a MERGE statement you can refer to columns from both the target and the source. For example, suppose that in the multi-row INSERT example from the INSERT with OUTPUT section you wanted the OUTPUT clause to return both the source key and the target key. This cannot be done directly in the INSERT statement because you don't have access to the source table. You can achieve this with the MERGE statement instead. Because an INSERT action is only allowed when the merge predicate is false, simply use a condition that is always false, such as 1 = 2. The revised example would be as follows:

```
MERGE INTO Sales.MyOrders AS TGT
USING ( SELECT orderid, custid, empid, orderdate
        FROM Sales.Orders
        WHERE shipcountry = N'Norway' ) AS SRC
  ON 1 = 2
WHEN NOT MATCHED THEN
  INSERT(custid, empid, orderdate) VALUES(custid, empid, orderdate)
OUTPUT
  SRC.orderid AS srcorderid, inserted.orderid AS tgtorderid,
  inserted.custid, inserted.empid, inserted.orderdate;
```

At this point, run the following code to clear the table and reset the sequence start value to 1:

```
TRUNCATE TABLE Sales.MyOrders;
ALTER SEQUENCE Sales.SeqOrderIDs RESTART WITH 1;
```

Nested DML

Suppose you need to capture output from a modification statement, but you are interested only in a subset of the output rows and not all of them. T-SQL has a solution for this in the form of *nested DML* (data manipulation language).

With T-SQL, you can define something that looks like a derived table based on a modification with an OUTPUT clause. Then you can have an outer INSERT SELECT statement against a target table, with the source table being this special derived table. The outer INSERT SELECT can have a WHERE clause that filters the output rows from the derived table, inserting only the rows that satisfy the search condition into the target. The outer INSERT SELECT statement cannot have other elements besides WHERE like table operators, GROUP BY, HAVING, and so on.

As an example of nested DML, consider the previous MERGE statement. Suppose that you need to capture only the rows affected by an INSERT action in a table variable for further processing. You can achieve this by using the following code:

```
DECLARE @InsertedOrders AS TABLE
(
  orderid   INT  NOT NULL PRIMARY KEY,
  custid    INT  NOT NULL,
  empid     INT  NOT NULL,
  orderdate DATE NOT NULL
);

INSERT INTO @InsertedOrders(orderid, custid, empid, orderdate)
  SELECT orderid, custid, empid, orderdate
  FROM (MERGE INTO Sales.MyOrders AS TGT
    USING (VALUES(1, 70, 1, '20151218'), (2, 70, 7, '20160429'), (3, 70, 7, '20160820'),
                 (4, 70, 3, '20170114'), (5, 70, 1, '20170226'), (6, 70, 2, '20170410'))
               AS SRC(orderid, custid, empid, orderdate)
          ON SRC.orderid = TGT.orderid
        WHEN MATCHED AND EXISTS( SELECT SRC.* EXCEPT SELECT TGT.* ) THEN
          UPDATE SET TGT.custid    = SRC.custid,
                     TGT.empid     = SRC.empid,
                     TGT.orderdate = SRC.orderdate
        WHEN NOT MATCHED THEN
          INSERT VALUES(SRC.orderid, SRC.custid, SRC.empid, SRC.orderdate)
        WHEN NOT MATCHED BY SOURCE THEN
          DELETE
        OUTPUT
          $action AS the_action, inserted.*) AS D
  WHERE the_action = 'INSERT';

SELECT * FROM @InsertedOrders;
```

Notice the derived table D that is defined based on the MERGE statement with the OUTPUT clause. The OUTPUT clause returns, among other things, the result of the $action function, naming the target column the_action. The code uses an INSERT SELECT statement with the source being the derived table D and the target table being the table variable @InsertedOrders. The WHERE clause in the outer query filters only the rows that have the INSERT action.

When you run the previous code for the first time, you get the following output:

```
orderid     custid      empid       orderdate
----------- ----------- ----------- ----------
1           70          1           2015-12-18
2           70          7           2016-04-29
3           70          7           2016-08-20
4           70          3           2017-01-14
5           70          1           2017-02-26
6           70          2           2017-04-10
```

Run it for the second time. It should return an empty set this time.

Impact of structural changes on data

This section describes the impact of structural changes like adding, dropping, and altering columns on data. In the examples in this section I use the Sales.MyOrders table and Sales.SeqOrderIDs sequence from the Merging data section. Run the following code to populate the table with initial sample data:

```
TRUNCATE TABLE Sales.MyOrders;
ALTER SEQUENCE Sales.SeqOrderIDs RESTART WITH 1;
INSERT INTO Sales.MyOrders(custid, empid, orderdate)
  VALUES(70, 1, '20151218'), (70, 7, '20160429'), (70, 7, '20160820'),
        (70, 3, '20170114'), (70, 1, '20170226'), (70, 2, '20170410');
```

Adding a column

In order to add a column to a table, you use the following syntax:

```
ALTER TABLE < table_name > ADD <column_definition> [<column_constraint>] [WITH VALUES];
```

If the table is empty, you can add a column that doesn't allow NULLs and also doesn't get its values somehow automatically. If the table isn't empty, such an attempt fails. To demonstrate this, run the following code:

```
ALTER TABLE Sales.MyOrders ADD requireddate DATE NOT NULL;
```

This attempt fails with the following error:

```
Msg 4901, Level 16, State 1, Line 608
ALTER TABLE only allows columns to be added that can contain nulls, or have a DEFAULT
definition specified, or the column being added is an identity or timestamp column, or
alternatively if none of the previous conditions are satisfied the table must be empty
to allow addition of this column. Column 'requireddate' cannot be added to non-empty
table 'MyOrders' because it does not satisfy these conditions.
```

Read the error message carefully. Observe that in order to add a column to a nonempty table, the column either needs to allow NULLs, or somehow get its values automatically. For instance, you can associate a default constraint with the column when you add it. You can also indicate that you want the default expression to be applied to the existing rows by adding the WITH VALUES clause as follows:

```
ALTER TABLE Sales.MyOrders
  ADD requireddate DATE NOT NULL
  CONSTRAINT DFT_MyOrders_requireddate DEFAULT ('19000101') WITH VALUES;
```

Note that if the column is defined as NOT NULL as in our case, the default expression is applied with or without this clause. If the column allows NULLs, without the clause a NULL is used and with the clause the default expression is used.

Query the table after adding this column and notice that the requireddate is January 1, 1900 in all rows.

Dropping a column

In order to drop a column from a table, you use the following syntax:

```
ALTER TABLE <table_name> DROP COLUMN <column_name>;
```

The attempt to drop the column fails when the column:

- Is used in an index.
- Is used in a default, check, foreign key, unique, or primary key constraint.
- Is bound to a default object or a rule.

For example, try to drop the requireddate column by running the following code:

```
ALTER TABLE Sales.MyOrders DROP COLUMN requireddate;
```

This attempt fails because there's a default constraint associated with the column. You get the following error:

```
Msg 5074, Level 16, State 1, Line 631
The object 'DFT_MyOrders_requireddate' is dependent on column 'requireddate'.
Msg 4922, Level 16, State 9, Line 631
ALTER TABLE DROP COLUMN requireddate failed because one or more objects access this
column.
```

In order to drop the column, you need to drop the constraint first.

Altering a column

In order to alter a column, you use the following syntax:

```
ALTER TABLE <table_name> ALTER COLUMN <column_definition> WITH ( ONLINE = ON | OFF );
```

There are a number of cases where the attempt to alter the column fails (partial list):

- When used in a primary key or foreign key constraint.
- When used in a check or unique constraint, unless you're just keeping or increasing the length of a variable-length column.
- When used in a default constraint, unless you're changing the length, precision, or scale of a column as long as the data type is not changed.

As an example, run the following code in attempt to change the data type of the required-date column from DATE to DATETIME:

```
ALTER TABLE Sales.MyOrders ALTER COLUMN requireddate DATETIME NOT NULL;
```

This attempts fails with the following error because there's a default constraint associated with the column.

```
Msg 5074, Level 16, State 1, Line 649
The object 'DFT_MyOrders_requireddate' is dependent on column 'requireddate'.
Msg 4922, Level 16, State 9, Line 649
ALTER TABLE ALTER COLUMN requireddate failed because one or more objects access this
column.
```

In terms of nullability, if a column is included in a primary key constraint you cannot alter it from NOT NULL to NULL, otherwise you can. For example, run the following code:

```
ALTER TABLE Sales.MyOrders ALTER COLUMN requireddate DATE NULL;
```

This code completes successfully.

As for altering a column that allows NULLs to not allowing NULLs, this is allowed as long as there are no NULLs present in the data. Run the following code:

```
ALTER TABLE Sales.MyOrders ALTER COLUMN requireddate DATE NOT NULL;
```

This code completes successfully.

If there are NULLs present in the data, an attempt to add the NOT NULL constraint fails. In a similar way, attempting to add a primary key or unique constraint fails if duplicates exist in the data. With check and foreign key constraints you do have control over whether existing data is verified or not. By default SQL Server uses a WITH CHECK mode that verifies that the existing data meets the constraint's requirements, and fails the attempt to add the constraint if the data is invalid. However, you can specify the WITH NOCHECK option to ask SQL Server not to verify existing data.

For many column alteration operations, SQL Server supports indicating the option ONLINE = ON (it is OFF by default). With this option set to ON, the table is available while the alter operation is in progress. Examples for operations that can be done online include a change in the data type, nullability, precision, length and others.

If you need to alter a column to start or stop getting its values from a sequence object you can achieve this easily by either adding or dropping a default constraint with the NEXT VALUE FOR function call. For instance, the orderid column in the Sales.MyOrders table gets its values from the Sales.SeqOrderIDs sequence using a default constraint. To drop the default constraint from the existing table, use the following code:

```
ALTER TABLE Sales.MyOrders DROP CONSTRAINT DFT_MyOrders_orderid;
```

To add the constraint, use the following code:

```
ALTER TABLE Sales.MyOrders ADD CONSTRAINT DFT_MyOrders_orderid
  DEFAULT(NEXT VALUE FOR Sales.SeqOrderIDs) FOR orderid;
```

With identity, it's not that simple. You're not allowed to alter a column to add or remove the identity property. So if you need to apply such a change, it's a very expensive offline operation that involves creating another table, copying the data, dropping the original table, and renaming the new table to the original table name.

EXAM TIP

Despite the fact that the exam tries to measure your real life knowledge and proficiency with the subject matter, keep in mind that the exam puts you in different conditions than in real life. For example, when you take the exam you don't have access to any online or offline resources, unlike in life. This means that you need to memorize the syntax of the different T-SQL statements that are covered by the exam. Also, try to focus on what the question is asking exactly, and what seems to be the most correct answer to the question, as opposed to what is considered the best practice or how you would have done things.

For more information about altering tables see the official documentation on the topic at *https://msdn.microsoft.com/en-us/library/ms190273.aspx*.

When you're done, run the following code for cleanup:

```
DROP TABLE IF EXISTS Sales.MyOrders;
DROP SEQUENCE IF EXISTS Sales.SeqOrderIDs;
```

Chapter summary

- Understanding the foundations of T-SQL is key to writing correct and robust code.
- Logical query processing describes the conceptual interpretation of a query, and evaluates the major query clauses in the following order: FROM, WHERE, GROUP BY, HAVING, SELECT, ORDER BY.
- Use the WHERE clause to filter data based on predicates and always remember to think of NULLs and the three-valued-logic (true, false and unknown).
- Use the ORDER BY clause in the outer query to apply presentation ordering to the query result, and remember that a query without an order by clause does not guarantee presentation order, despite any observed behavior.
- Use the UNION, UNION ALL, INTERSECT and EXCEPT operators to combine query results. These operators use distinctness-based comparison unlike the WHERE, ON and HAVING clauses, which use equality-based comparison.
- Joins allow you to combine rows from tables and return both matched attributes and additional attributes from both sides.
- Cross joins return a Cartesian product of the two inputs.
- Inner joins return only matching rows from the two sides.
- Outer joins return both matching rows and rows without matches from the nonpreserved side or sides.
- In an outer join the ON clauses serves a matching purpose and the WHERE clause a filtering purpose.

- If NULLs are present in the join columns you need to add special handling. Avoid applying manipulation on the join columns to preserve the data's ordering property and allow efficient use of indexing.
- T-SQL provides you with built-in functions of various categories such as string, date and time, conversion, system, and others.
- Scalar-valued functions return a single value; table-valued functions return a table result and are used in the FROM clause of a query. Aggregate functions are applied to a set and return a single value, and can be used in grouped queries and windowed queries.
- When at all possible, try to avoid applying manipulation to filtered columns to enable filter sargability and efficient use of indexes.
- Function determinism determines whether the function is guaranteed to return the same output given the same set of inputs.
- T-SQL supports the following statements to modify data: INSERT, SELECT INTO, UPDATE, DELETE, TRUNCATE TABLE, and MERGE.
- Use the OUTPUT clause in a modification statement to return data from the modified rows for purposes like auditing, archiving, and others. You can either return the result set to the caller, or write it to a table using the INTO clause.
- Make sure you understand the impact that structural changes to a table like adding, altering and dropping columns have on existing data.

Thought experiment

In this thought experiment, demonstrate your skills and knowledge of the topics covered in this chapter. You can find the answer to this thought experiment in the next section.

You're being interviewed for a T-SQL developer role in the IT department of a large technology company. Answer the following questions to the best of your knowledge:

1. How come you cannot use an alias you define in the SELECT list in the WHERE clause, or even the same SELECT clause? Where can you use such an alias?
2. What are the differences between joins and set operators?
3. What could prevent SQL Server from treating a query filter optimally, meaning, from using an index efficiently to support the filter? What other query elements could also be affected in a similar manner and what can you do to get optimal treatment?
4. What is the difference between the ON and WHERE clauses?
5. Explain what function determinism means and what are the implications of using nondeterministic functions?
6. What are the differences between DELETE and TRUNCATE TABLE?

7. You need to perform a multi-row insert into a target table that has a column with an identity property. You need to capture the newly generated identity values for further processing. How can you achieve this?
8. When should you use the WITH VALUES clause explicitly as part of adding a column to a table?

Thought experiment answer

This section contains the solution to the thought experiment.

1. According to logical query processing, which describes the conceptual interpretation of a query, the order in which the major query clauses are interpreted is: FROM, WHERE, GROUP BY, HAVING, SELECT, ORDER BY. You cannot use an alias that is created in the SELECT clause in clauses that are processed in an earlier step. This explains why you cannot use such an alias in the FROM, WHERE, GROUP BY and HAVING clauses. As for why you cannot use such an alias in other expressions in the same SELECT clause, that's because all expressions that appear in the same logical query-processing step are treated as a set, and a set has no order. The only clause that can refer to aliases that are created in the SELECT clause is the ORDER BY clause, because that's the only clause that is evaluated after the SELECT clause.
2. A join can compare a subset of the elements from the input tables while returning elements that it doesn't compare. Also, a join uses equality (or inequality) based comparison as the join predicate, whereas a comparison between two NULLs or between a NULL and anything yields unknown. A set operator implicitly compares all expressions in corresponding positions in the two input queries. Also, a set operator uses distinctness-based comparison, whereas a comparison between two NULLs yields true, and a comparison between a NULL and a non-NULL value yields false.
3. Manipulation of the filtered column in most cases prevents the filter's sargability. This means that the optimizer cannot rely on index order, for instance, to perform a seek within the index. In a similar way, manipulation of a column can prevent the optimizer from relying on index order for purposes of joining, grouping, and ordering.
4. In an outer join the ON clause serves a matching purpose. It determines which rows from the preserved side get matched with which rows from the non-preserved side. It cannot determine which rows from the preserved side of the join are returned—it's predetermined that all of them are returned. The WHERE clause serves a simpler filtering meaning. It determines which rows from the result of the FROM clause to keep and which to discard. In an inner join both ON and WHERE serve the same simple filtering meaning.
5. A function is said to be deterministic if given the same set of input values it is guaranteed to return repeatable results, otherwise it is said to be nondeterministic. If you use a nondeterministic function in a computed column, you cannot create an index on that

column. Similarly, if you use a nondeterministic function in a view, you cannot create a clustered index on the view.

6. DELETE supports a filter, is fully logged, and does not reset the current identity value. TRUNCATE TABLE has no filter, is minimally logged and therefore much faster than DELETE, and does reset the current identity value. Unlike DELETE, TRUNCATE TABLE is disallowed if there's an indexed view based on the table, or a foreign key pointing to the table, even if there are no related rows in the referencing table.
7. Use the OUTPUT clause and write the newly generated identity values along with any other data that you need from the inserted rows aside, for example into a table variable. You can then use the data from the table variable in the next step where you apply further processing.
8. When the column is defined as a nullable one, and you want to apply the default expression that is associated with the column in the new rows, you need to specify the WITH VALUES clause explicitly. If the column is defined as NOT NULL, and you associate a default expression with it, the default expression is applied even when not specifying the WITH VALUES clause explicitly.

This page intentionally left blank

CHAPTER 2

Query data with advanced Transact-SQL components

This chapter covers a number of T-SQL components that allow you to manipulate and analyze data, some of which might be considered more advanced than the ones covered in Chapter 1, "Manage data with Transact-SQL." You will learn how to nest queries and use the APPLY operator, work with table expressions, apply data analysis calculations with grouping, pivoting, and windowing, query historical data from temporal tables, and query and output XML and JSON data.

Skills in this chapter:

- Query data by using subqueries and APPLY
- Query data by using table expressions
- Group and pivot data by using queries
- Query temporal data and non-relational data

Skill 2.1: Query data by using subqueries and APPLY

This skill focuses on the nesting of queries, known as subqueries, and the APPLY operator, which allows you to apply a table subquery to each row from some table.

This section covers how to:

- Determine the results of queries using subqueries and table joins
- Evaluate performance differences between table joins and correlated subqueries based on provided data and query plans
- Distinguish between the use of CROSS APPLY and OUTER APPLY
- Write APPLY statements that return a given data set based on supplied data

Subqueries

Subqueries can be self-contained—independent of the outer query; or they can be correlated—namely, having a reference to a column from the table in the outer query. In terms of the result of the subquery, it can be scalar, multi-valued (table with a single column), or multi-column table-valued (table with multiple columns).

This section starts by covering the simpler self-contained subqueries, and then continues to correlated subqueries.

Self-contained subqueries

Self-contained subqueries are subqueries that have no dependency on the outer query. If you want, you can highlight the inner query in SSMS and run it independently. This makes the troubleshooting of problems with self-contained subqueries easier compared to correlated subqueries.

As mentioned, a subquery can return different forms of results. It can return a single value, table with multiple values in a single column, or even a multi-column table result. Table-valued subqueries, or table expressions, are discussed in Skill 2.2 later in this chapter.

Subqueries that return a single value, or scalar subqueries, can be used where a single-valued expression is expected, like in one side of a comparison. For example, the following query uses a self-contained subquery to return the products with the minimum unit price:

```
USE TSQLV4;

SELECT productid, productname, unitprice
FROM Production.Products
WHERE unitprice =
  (SELECT MIN(unitprice)
   FROM Production.Products);
```

Here's the output of this query.

```
productid  productname    unitprice
---------- -------------- ----------
33         Product ASTMN  2.50
```

As you can see, the subquery returns the minimum unit price from the Production.Products table. The outer query then returns information about products with the minimum unit price. Try highlighting only the inner query and executing it, and you will find that this is possible.

Note that if what's supposed to be a scalar subquery returns in practice more than one value, the code fails at run time. If the scalar subquery returns an empty set, it is converted to a NULL.

A subquery can also return multiple values in the form of a single column and multiple rows. Such a subquery can be used where a multi-valued result is expected—for example,

when using the IN predicate. As an example, the following query uses a multi-valued subquery to return products supplied by suppliers from Japan.

```
SELECT productid, productname, unitprice
FROM Production.Products
WHERE supplierid IN
  (SELECT supplierid
   FROM Production.Suppliers
   WHERE country = N'Japan');
```

This query generates the following output.

```
productid  productname    unitprice
---------- -------------- ----------
9          Product AOZBW  97.00
10         Product YHXGE  31.00
13         Product POXFU  6.00
14         Product PWCJB  23.25
15         Product KSZOI  15.50
74         Product BKAZJ  10.00
```

The inner query returns supplier IDs of suppliers from Japan. The outer query then returns information about products whose supplier ID is in the set returned by the subquery. As with predicates in general, you can negate an IN predicate, so if you wanted to return products supplied by suppliers that are not from Japan, simply change IN to NOT IN.

T-SQL supports a few esoteric predicates that operate on subqueries. Those are ALL, ANY and SOME. They are rarely used because there are usually simpler and more intuitive alternatives, but since there's a chance that they will be mentioned in the exam, you want to make sure that you are familiar with them. Following is the form for using these elements:

```
SELECT <select_list>
FROM <table>
WHERE <expression> <operator> {ALL | ANY | SOME} (<subquery>);
```

The ALL predicate returns true only if when applying the operator to the input expression and all values returned by the subquery, you get a true in all cases. For example, the following query is an alternative solution to the one shown earlier for returning the product with the minimum unit price:

```
SELECT productid, productname, unitprice
FROM Production.Products
WHERE unitprice <= ALL (SELECT unitprice FROM Production.Products);
```

The way the query is phrased is "return the products where the unit price is less than or equal to all product unit prices."

The ANY and SOME predicates have identical meaning. Suffice that you get a true for at least one of the values returned by the subquery for the whole predicate to return true. As an example, the following query returns all products with a unit price that is greater than the minimum.

```
SELECT productid, productname, unitprice
FROM Production.Products
WHERE unitprice > ANY (SELECT unitprice FROM Production.Products);
```

The way the query is phrased is "return the products where the unit price is greater than any product unit prices." This will be false only for the product with the minimum price.

Correlated subqueries

Correlated subqueries are subqueries where the inner query has a reference to a column from the table in the outer query. They are trickier to work with compared to self-contained subqueries because you can't just highlight the inner portion and run it independently.

As an example, suppose that you need to return products with the minimum unit price per category. You can use a correlated subquery to return the minimum unit price out of the products where the category ID is equal to the one in the outer row (the correlation), as follows:

```
SELECT categoryid, productid, productname, unitprice
FROM Production.Products AS P1
WHERE unitprice =
  (SELECT MIN(unitprice)
   FROM Production.Products AS P2
   WHERE P2.categoryid = P1.categoryid);
```

This query generates the following output:

```
categoryid  productid  productname    unitprice
----------- ---------- -------------- ----------
1           24         Product QOGNU  4.50
2           3          Product IMEHJ  10.00
3           19         Product XKXDO  9.20
4           33         Product ASTMN  2.50
5           52         Product QSRXF  7.00
6           54         Product QAQRL  7.45
7           74         Product BKAZJ  10.00
8           13         Product POXFU  6.00
```

Notice that the outer query and the inner query refer to different instances of the same table, Production.Products. In order for the subquery to be able to distinguish between the two, you must assign different aliases to the different instances. The query assigns the alias P1 to the outer instance and P2 to the inner instance, and by using the table alias as a prefix, you can refer to columns in an unambiguous way. The subquery uses a correlation in the predicate `P2.categoryid = P1.categoryid`, meaning that it filters only the products where the category ID is equal to the one in the outer row. So, when the outer row has category ID 1, the inner query returns the minimum unit price out of all products where the category ID is 1. And when the outer row has category ID 2, the inner query returns the minimum unit price out of all the products where the category ID is 2; and so on.

As another example of a correlated subquery, the following query returns customers who placed orders on February 12, 2016:

```
SELECT custid, companyname
FROM Sales.Customers AS C
WHERE EXISTS
  (SELECT *
   FROM Sales.Orders AS O
   WHERE O.custid = C.custid
     AND O.orderdate = '20070212');
```

This query generates the following output:

```
custid  companyname
------- ---------------
45      Customer QXPPT
48      Customer DVFMB
76      Customer SFOGW
```

The EXISTS predicate accepts a subquery as input and returns true when the subquery returns at least one row and false otherwise. In this case, the subquery returns orders placed by the customer whose ID is equal to the customer ID in the outer row (the correlation) and where the order date is February 12, 2016. So the outer query returns a customer only if there's at least one order placed by that customer on the date in question.

As a predicate, EXISTS doesn't need to return the result set of the subquery; rather, it returns only true or false, depending on whether the subquery returns any rows. For this reason, the query optimizer ignores the SELECT list of the subquery, and therefore, whatever you specify there will not affect optimization choices like index selection.

As with other predicates, you can negate the EXISTS predicate as well. The following query negates the previous query's predicate, returning customers who did not place orders on February 12, 2016:

```
SELECT custid, companyname
FROM Sales.Customers AS C
WHERE NOT EXISTS
  (SELECT *
   FROM Sales.Orders AS O
   WHERE O.custid = C.custid
     AND O.orderdate = '20160212');
```

This query generates the following output, shown here in abbreviated form:

```
custid  companyname
------- ---------------
72      Customer AHPOP
58      Customer AHXHT
25      Customer AZJED
18      Customer BSVAR
91      Customer CCFIZ
...
```

Optimization of subqueries versus joins

When comparing the performance of solutions using subqueries versus solutions using joins, you will find that it's not like one tool always performs better than the other. There are cases where you will get the same query execution plans for both, cases where subqueries perform better, and cases where joins perform better. Ultimately, in performance critical cases you will want to test solutions based on both tools. However, there are specific aspects of these tools where SQL Server is known to handle one better than the other that you want to make sure that you're aware of.

I'll start with an example where subqueries are optimized less efficiently than joins. If you have multiple subqueries that need to apply computations such as aggregates based on the same set of rows, SQL Server will perform a separate access to the data for each subquery. With a join, you can apply multiple aggregate calculations based on the same access to the data. For example, suppose that you need to query the Sales.Orders table and compute for each order the percent of the current freight value out of the customer total, as well as the difference from the customer average. You create the following covering index to support your solutions:

```
CREATE INDEX idx_cid_i_frt_oid
  ON Sales.Orders(custid) INCLUDE(freight, orderid);
```

Here's the solution for the task using correlated subqueries:

```
SELECT orderid, custid, freight,
  freight / ( SELECT SUM(O2.freight)
              FROM Sales.Orders AS O2
              WHERE O2.custid = O1.custid ) AS pctcust,
  freight - ( SELECT AVG(O3.freight)
              FROM Sales.Orders AS O3
              WHERE O3.custid = O1.custid ) AS diffavgcust
FROM Sales.Orders AS O1;
```

Here's the solution for the task using a derived table and a join:

```
SELECT O.orderid, O.custid, O.freight,
  freight / totalfreight AS pctcust,
  freight - avgfreight AS diffavgcust
FROM Sales.Orders AS O
  INNER JOIN ( SELECT custid, SUM(freight) AS totalfreight, AVG(freight) AS avgfreight
               FROM Sales.Orders
               GROUP BY custid ) AS A
    ON O.custid = A.custid;
```

The query in the derived table A (more on derived tables in Skill 2.2) computes the customer aggregates, and the outer query joins the detail and the aggregates based on a match between the customer IDs and computes the percent and difference from the average.

Figure 2-1 shows the execution plans for both solutions. Query 1 represents the solution with the subqueries and Query 2 represents the solution with the join.

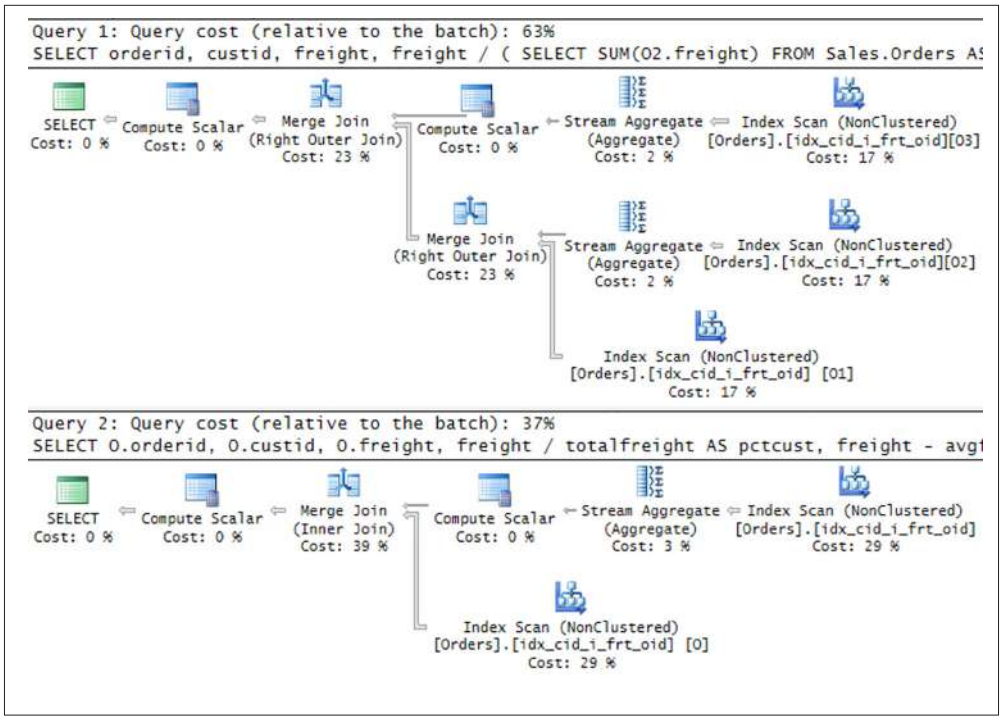


FIGURE 2-1 Query plans for solutions that compute percent of total and difference from the average

Observe in the first plan that the index is accessed three times; once for the detail reference to the table (instance O1) and two additional times for the two subqueries (instances O2 and O3). In the second plan the index is accessed only twice; once for the detail reference (instance O), and only one more time for the computation of both aggregates. Also notice the relative cost of each query plan out of the entire batch; the first plan costs twice as much as the second.

When you're done, run the following code for cleanup:

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

In the second example, consider a case where SQL Server optimizes subqueries better than joins. For this example, first run the following code to add a shipper row into the Sales.Shippers table:

```
INSERT INTO Sales.Shippers(companyname, phone)
  VALUES('Shipper XYZ', '(123) 456-7890');
```

Your task is to write a solution that returns shippers who didn't handle any orders yet. The important index for this task is a nonclustered index on the shipperid column in the Sales.Orders table, which already exists.

Here's a solution to the task based on a subquery.

```
SELECT S.shipperid
FROM Sales.Shippers AS S
WHERE NOT EXISTS
  (SELECT *
   FROM Sales.Orders AS O
   WHERE O.shipperid = S.shipperid);
```

Here's a solution to the task based on a join:

```
SELECT S.shipperid
FROM Sales.Shippers AS S
  LEFT OUTER JOIN Sales.Orders AS O
    ON S.shipperid = O.shipperid
WHERE O.orderid IS NULL;
```

Figure 2-2 shows the execution plans for both queries. Query 1 represents the solution based on the subquery and Query 2 represents the solution based on the join.

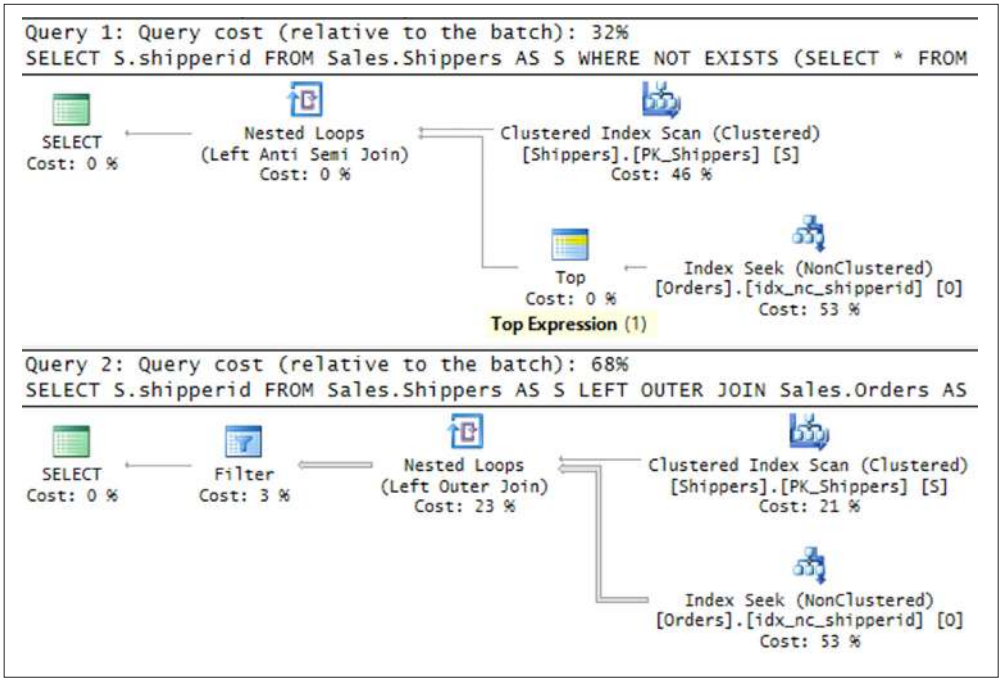


FIGURE 2-2 Query plans for solutions that identify shippers without orders

Both plans use a Nested Loops algorithm to process the join. In this algorithm the outer input of the loop scans shipper rows from the Sales.Shippers table. For each shipper row, the inner input of the loop looks for matching orders in the nonclustered index on Sales.Orders. The key difference between the plans is that with the subquery-based solution the optimizer is capable of using a specialized optimization called Anti Semi Join. With this optimization, as soon as a match is found, the execution short circuits (notice the Top operator with the Top Expression property set to 1). With the join-based solution a shipper row is matched with all

of its corresponding orders, and later the plan filters only the shippers that didn't have any matches. Observe the relative cost of each query plan out of the entire batch. The plan for the subquery-based solution costs less than half of the plan for the join-based solution. At the time of writing, SQL Server does not use the Anti Semi Join optimization for queries based on an actual join, but does so for queries based on subqueries and set operators.

When you're done, run the following code to delete the new shipper row:

```
DELETE FROM Sales.Shippers WHERE shipperid > 3;
```

In short, when optimizing your solutions, it's important to be informed about cases where one tool does better than another. Also, make sure to keep an open mind, test different solutions, compare their run times and query plans, and eventually choose the optimal one.

The APPLY operator

The APPLY operator is a powerful operator that you can use to apply some query logic to each row from a table. The operator evaluates the left input first, and for each of its rows, applies a derived table query or table function that you provide as the right input. Let's refer to the right input in short as a table expression (more on table expressions in Skill 2.2). What's interesting about the APPLY operator as compared to a join is that a join treats its two inputs as a set of inputs, and recall that a set has no order. This means that if any of the join inputs is a query, you cannot refer in that query to elements from the other side. In other words—correlations aren't allowed. Conversely, the APPLY operator evaluates the left side first, and for each of the left rows, applies the table expression that you provide as the right input. As a result, the query in the right side can have references to elements from the left side. If this sounds similar to a correlated subquery, that's for a good reason. The references from the right side to elements from the left are correlations. However, with normal subqueries you're generally limited to returning only one column, whereas with an applied table expression you can return a whole table result with multiple columns and multiple rows. This means that you can replace the use of cursors in some cases with the APPLY operator.

For example, suppose that you have a query that performs some logic for a particular supplier. And let's also suppose that you need to apply this query logic to each supplier from the Production.Suppliers table. You could use a cursor to iterate through the suppliers, and in each iteration invoke the query for the current supplier. Instead, you can use the APPLY operator, providing the Production.Suppliers table as the left input, and a table expression based on your query as the right input. You can correlate the supplier ID in the inner query of the right table expression to the supplier ID from the left table.

The two forms of the APPLY operator—CROSS and OUTER—are described in the next sections.

Before running the code samples in these sections, add a row to the Suppliers table by running the following code:

```
INSERT INTO Production.Suppliers
```

```
(companyname, contactname, contacttitle, address, city, postalcode, country, phone)
VALUES(N'Supplier XYZ', N'Jiru', N'Head of Security', N'42 Sekimai Musashino-shi',
       N'Tokyo', N'01759', N'Japan', N'(02) 4311-2609');
```

CROSS APPLY

The CROSS APPLY operator operates on left and right inputs. The right table expression can have a correlation to elements from the left table. The right table expression is applied to each row from the left input. What's special about the CROSS APPLY operator as compared to OUTER APPLY is that if the right table expression returns an empty set for a left row, the left row isn't returned. The reason that this operator is called *CROSS* APPLY is that per the left row, the operator behaves like a cross join between that row and the result set returned for it from the right input. Figure 2-3 shows an illustration of the CROSS APPLY operator.

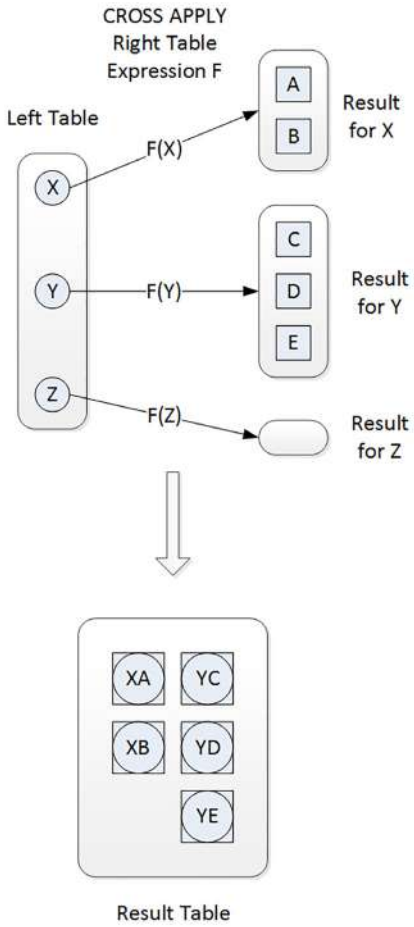


FIGURE 2-3 The CROSS APPLY operator

The letters X, Y, and Z represent key values from the left table. F represents the table expression provided as the right input, and in parentheses, you can see the key value from the left row passed as the correlated element. On the right side of the illustration, you can see the result returned from the right table expression for each left row. Then at the bottom, you can see the result of the CROSS APPLY operator, where left rows are matched with the respective right rows that were returned for them. Notice that a left row that gets an empty set back from the right table expression isn't returned. That's just like with a cross join between one row and zero rows; the result is an empty set. Such is the case with the row with the key value Z.

As an example, consider the following query, which returns the two products with the lowest unit prices for supplier 1:

```
SELECT TOP (2) productid, productname, unitprice
FROM Production.Products
WHERE supplierid = 1
ORDER BY unitprice, productid;
```

This query generates the following output.

```
productid  productname    unitprice
---------- -------------- ----------
3          Product IMEHJ  10.00
1          Product HHYDP  18.00
```

Suppose that you need to apply this logic to each of the suppliers from Japan that you have in the Production.Suppliers table. You don't want to use a cursor to iterate through the suppliers one at a time and invoke a separate query for each. Instead, you can use the CROSS APPLY operator, like so:

```
SELECT S.supplierid, S.companyname AS supplier, A.*
FROM Production.Suppliers AS S
  CROSS APPLY (SELECT TOP (2) productid, productname, unitprice
               FROM Production.Products AS P
               WHERE P.supplierid = S.supplierid
               ORDER BY unitprice, productid) AS A
WHERE S.country = N'Japan';
```

This query generates the following output.

```
supplierid  supplier        productid  productname    unitprice
----------- --------------- ---------- -------------- ----------
4           Supplier QOVFD  74         Product BKAZJ  10.00
4           Supplier QOVFD  10         Product YHXGE  31.00
6           Supplier QWUSF  13         Product POXFU  6.00
6           Supplier QWUSF  15         Product KSZOI  15.50
```

As you can see in the query, the left input to the APPLY operator is the Production.Suppliers table, with only suppliers from Japan filtered. The right table expression is a correlated derived table returning the two products with the lowest prices for the left supplier. Because the APPLY operator applies the right table expression to each supplier from the left, you get the two products with the lowest prices for each supplier from Japan. Because the CROSS

APPLY operator doesn't return left rows for which the right table expression returns an empty set, suppliers from Japan who don't have any related products aren't returned.

OUTER APPLY

The OUTER APPLY operator extends what the CROSS APPLY operator does by also including in the result rows from the left side that get an empty set back from the right side. NULLs are used as placeholders for the result columns from the right side. In other words, the OUTER APPLY operator preserves the left side. In a sense, for each single left row, the difference between OUTER APPLY and CROSS APPLY is similar to the difference between a LEFT OUTER JOIN and a CROSS JOIN. Figure 2-4 shows an illustration of the OUTER APPLY operator:

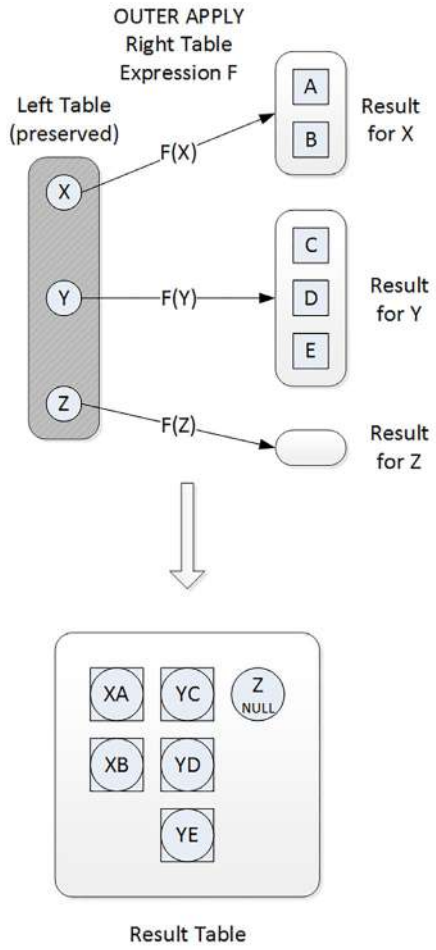


FIGURE 2-4 The OUTER APPLY operator

Observe that this time the left row with the key value Z is preserved.

Let's re-examine the example returning the two products with the lowest prices per supplier from Japan: If you use the OUTER APPLY operator instead of CROSS APPLY, you will preserve the left side. Here's the revised query.

```
SELECT S.supplierid, S.companyname AS supplier, A.*
FROM Production.Suppliers AS S
  OUTER APPLY (SELECT TOP (2) productid, productname, unitprice
               FROM Production.Products AS P
               WHERE P.supplierid = S.supplierid
               ORDER BY unitprice, productid) AS A
WHERE S.country = N'Japan';
```

Here's the output of this query.

```
supplierid  supplier        productid  productname    unitprice
----------- --------------- ---------- -------------- ----------
4           Supplier QOVFD  74         Product BKAZJ  10.00
4           Supplier QOVFD  10         Product YHXGE  31.00
6           Supplier QWUSF  13         Product POXFU  6.00
6           Supplier QWUSF  15         Product KSZOI  15.50
30          Supplier XYZ    NULL       NULL           NULL
```

Observe that supplier 30 was preserved this time even though it has no related products.

When you're done, run the following code to delete the supplier row that you added earlier:

```
DELETE FROM Production.Suppliers WHERE supplierid > 29;
```

MORE INFO **ON APPLY**

For more information and examples of creative uses of the APPLY operator, make sure to watch the Microsoft Virtual Academy video seminar "Boost your T-SQL with the APPLY operator" at *http://aka.ms/BoostTSQL* or at *https://channel9.msdn.com/Series/Boost-Your-T-SQL-With-the-APPLY-Operator*.

MORE INFO **ON LOGICAL ASPECTS OF APPLY**

For details about the logical aspects of APPLY, see "Logical Query Processing: The FROM Clause and APPLY" at *http://sqlmag.com/sql-server/logical-query-processing-clause-and-apply*.

Skill 2.2: Query data by using table expressions

This skill focuses on querying data by using table expressions. It starts with a description of what table expressions are, compares them to temporary tables, and then provides the details about the different kinds of table expressions.

This section covers how to:

- Identify basic components of table expressions
- Define usage differences between table expressions and temporary tables
- Construct recursive table expressions to meet business requirements

Table expressions, described

Table expressions are named queries. You write an inner query that returns a relational result set, name it, and query it from an outer query. T-SQL supports four forms of table expressions: derived tables, common table expressions (CTEs), views and inline table-valued functions.

The first two forms are visible only to the statement that defines them. As for the last two forms, you preserve the definition of the table expression in the database as an object; therefore, it's reusable, and you can also control access to the object with permissions.

Note that because a table expression is supposed to represent a relation, the inner query defining it needs to be relational. This means that all columns returned by the inner query must have names (use aliases if the column is a result of an expression), and all column names must be unique. Also, the inner query is not allowed to have an ORDER BY clause. (Remember, a set has no order.) There's an exception to the last rule: If you use the TOP or OFFSET-FETCH option in the inner query, the ORDER BY serves a meaning that is not related to presentation ordering; rather, it's part of the filter's specification. So if the inner query uses the TOP or OFFSET-FETCH option, it's allowed to have an ORDER BY clause as well. But then the outer query has no presentation ordering guarantees if it doesn't have its own ORDER BY clause.

Table expressions or temporary tables?

It's important to note that, from a performance standpoint, when SQL Server optimizes queries involving table expressions, it first unnests, or inlines, the table expression's logic, and therefore interacts with the underlying tables directly. It does not somehow persist the table expression's result in an internal work table and then interact with that work table. If for optimization reasons you do need to persist the result of a query for further processing, you should be using a temporary table or table variable.

There are cases where the use of table expressions is more optimal than temporary tables. For instance, imagine that you need to query some table T1 only once, then interact with the result of that query from some outer query, and finally interact with the result of that outer query from yet another outer query. You do not want to pay the penalty of writing the intermediate results physically to some temporary table, rather, you want the physical processing to interact directly with the underlying table. To achieve this, define a table expression based on the query against T1, give it a name, say D1, and then write an outer query against D1. Behind the scenes, SQL Server will unnest, or inline, the logic of the inner queries, like pealing the layers of an onion, and the query plan will interact directly with T1.

There are cases where you will get more optimal treatment when using temporary tables (which you create like regular tables, and name with a # sign as a prefix, such as #T1) or table variables (which you declare, and name with the @ sign as a prefix, such as @T1). That's typically the case when you have some expensive query, like one that scans large tables, joins them, and groups and aggregates the data. You need to interact with that query result multiple times—whether with a single query that joins multiple instances of the result or with multiple separate queries. If you use a table expression, the physical treatment repeats the work for each reference. In such cases you want to persist the result of the expensive work in a temporary table or table variable, and then interact with that temporary object a number of times. Between table variables and temporary tables, the main difference from an optimization perspective is that SQL Server maintains full blown statistics on temporary tables but very minimal statistics on table variables. Therefore, cardinality estimates (estimates for row counts during optimization) tend to be more accurate with temporary tables. So, when dealing with very small amounts of data like just a few rows, typically it's recommended to use table variables since that's the assumption that the optimizer makes any way. With larger table sizes, the recommendation is to use temporary tables, to allow better estimates, that will hopefully result in more optimal plans.

The following sections describe the different forms of table expressions that T-SQL supports.

Derived tables

A derived table is a named table subquery. You define the derived table's inner query in parentheses in the FROM clause of the outer query, and specify the name of the derived table after the parentheses.

Before demonstrating the use of derived tables, this section describes a query that returns a certain desired result. Then it explains a need that cannot be addressed directly in the query, and shows how you can address that need by using a derived table (or any other table expression type for that matter).

Consider the following query, which computes row numbers for products, partitioned by `categoryid`, and ordered by `unitprice` and `productid`:

```
USE TSQLV4;

SELECT
  ROW_NUMBER() OVER(PARTITION BY categoryid
                    ORDER BY unitprice, productid) AS rownum,
  categoryid, productid, productname, unitprice
FROM Production.Products;
```

This query generates the following output, shown here in abbreviated form:

```
rownum  categoryid  productid  productname    unitprice
------- ----------- ---------- -------------- ----------
1       1           24         Product QOGNU  4.50
2       1           75         Product BWRLG  7.75
```

```
3       1           34          Product SWNJY  14.00
4       1           67          Product XLXQF  14.00
5       1           70          Product TOONT  15.00
...
1       2           3           Product IMEHJ  10.00
2       2           77          Product LUNZZ  13.00
3       2           15          Product KSZOI  15.50
4       2           66          Product LQMGN  17.00
5       2           44          Product VJIEO  19.45
...
```

You learn about the ROW_NUMBER function, as well as other window functions, in Skill 2.3; but for now, suffice it to say that the ROW_NUMBER function computes unique incrementing integers from 1 and is based on indicated ordering, possibly within partitions of rows. As you can see in the query's result, the ROW_NUMBER function generates unique incrementing integers starting with 1 based on `unitprice` and `productid` ordering, within each partition defined by `categoryid`.

The thing with the ROW_NUMBER function—and window functions in general—is that they are only allowed in the SELECT and ORDER BY clauses of a query. So, what if you want to filter rows based on such a function's result? For example, suppose you want to return only the rows where the row number is less than or equal to 2; namely, in each category you want to return the two products with the lowest unit prices, with the product ID used as a tiebreaker. You are not allowed to refer to the ROW_NUMBER function in the query's WHERE clause. Remember also that according to logical query processing, you're not allowed to refer to a column alias that was assigned in the SELECT list in the WHERE clause, because the WHERE clause is conceptually evaluated before the SELECT clause.

You can circumvent the restriction by using a table expression. You write a query such as the previous query that computes the window function in the SELECT clause, and assign a column alias to the result column. You then define a table expression based on that query, and refer to the column alias in the outer query's WHERE clause, like so:

```
SELECT categoryid, productid, productname, unitprice
FROM (SELECT
        ROW_NUMBER() OVER(PARTITION BY categoryid
                          ORDER BY unitprice, productid) AS rownum,
        categoryid, productid, productname, unitprice
      FROM Production.Products) AS D
WHERE rownum <= 2;
```

This query generates the following output, shown here in abbreviated form:

```
categoryid  productid  productname    unitprice
----------- ---------- -------------- ----------
1           24         Product QOGNU  4.50
1           75         Product BWRLG  7.75
2           3          Product IMEHJ  10.00
2           77         Product LUNZZ  13.00
3           19         Product XKXDO  9.20
3           47         Product EZZPR  9.50
...
```

As you can see, the derived table is defined in the FROM clause of the outer query in parentheses, followed by the derived table name. Then the outer query is allowed to refer to column aliases that were assigned by the inner query. That's a classic use of table expressions.

EXAM TIP

During the exam pay attention to multi-choice questions where some of the suggested answers include invalid code. For instance, code that defines an alias in the SELECT list and refers to that alias in clauses that are evaluated before the SELECT, like the WHERE clause, or even the same SELECT clause itself. Make sure that you identify the answer with the valid syntax after eliminating the ones that are invalid.

Two column aliasing options are available to you when working with derived tables: both inline and external. With the inline form, you specify the column alias as part of the expression, as in <expression> AS alias. The last query used the inline form to assign the alias `rownum` to the expression with the ROW_NUMBER function. With the external aliasing form, you don't specify result column aliases as part of the column expressions; instead, you name all target columns right after the derived table's name, as in FROM (...) AS D(`rownum`, `categoryid`, `productid`, `productname`, `unitprice`). With the external form, you must specify all target column names and not just those that are results of computations. If you use both inline and external aliases, the external ones prevail as far as the outer query is concerned.

There are a couple of problematic aspects to working with derived tables that stem from the fact that a derived table is defined in the FROM clause of the outer query. One problem has to do with cases where you want to refer to one derived table from another. In such a case, you end up nesting derived tables, and nesting often complicates the logic, making it hard to follow and increasing the likelihood for errors. Consider the following general form of nesting of derived tables:

```
SELECT ...
FROM (SELECT
      FROM (SELECT ...
            FROM T1
            WHERE ...) AS D1
      WHERE ...) AS D2
WHERE ...;
```

The other problem with derived tables has to do with the fact that a join treats its two inputs as a set, meaning no order; the two inputs are evaluated in an *all-at-once* manner. As a result, if you define a derived table as the left input of the join, that derived table is not visible to the right input of the join. This means that if you want to join multiple instances of the same derived table, you can't. You have no choice but to duplicate the code, defining multiple derived tables based on the same query. The general form of such a query looks like this:

```
SELECT ...
FROM (SELECT ...
      FROM T1) AS D1
  INNER JOIN
```

```
    (SELECT ...
     FROM T1) AS D2
  ON ...;
```

The derived tables D1 and D2 are based on the same query. This repetition of code increases the likelihood for errors when you need to make revisions to the inner queries.

Common table expressions

A common table expression (CTE) is a similar concept to a derived table in the sense that it's a named table expression that is visible only to the statement that defines it. Like a query against a derived table, a query against a CTE involves three main parts:

- The inner query
- The name you assign to the query and its columns
- The outer query

However, with CTEs, the arrangement of the three parts is different. Recall that with derived tables the inner query appears in the FROM clause of the outer query—in the middle of things. With CTEs, you first name the CTE, then specify the inner query, and then the outer query—a much more modular approach:

```
WITH <CTE_name>
AS
(
  <inner_query>
)
<outer_query>;
```

Recall the example from the section about derived tables where you returned for each product category the two products with the lowest unit prices. Here's how you can implement the same task with a CTE:

```
WITH C AS
(
  SELECT
    ROW_NUMBER() OVER(PARTITION BY categoryid
                      ORDER BY unitprice, productid) AS rownum,
    categoryid, productid, productname, unitprice
  FROM Production.Products
)
SELECT categoryid, productid, productname, unitprice
FROM C
WHERE rownum <= 2;
```

As you can see, it's a similar concept to derived tables, except the inner query is not defined in the middle of the outer query. Instead, first you name the table expression, then define the inner query—from start to end andthen the outer query—from start to end. This design leads to much clearer code that is easier to understand.

You don't nest CTEs like you do derived tables. If you need to define multiple CTEs, you simply separate them by commas. Each can refer to the previously defined CTEs, and the

outer query can refer to all of them. After the outer query terminates, all CTEs defined in that WITH statement are gone. The fact that you don't nest CTEs makes it easier to follow the logic and therefore reduces the chances for errors. For example, if you want to refer to one CTE from another, you can use the following general form:

```
WITH C1 AS
(
  SELECT ...
  FROM T1
  WHERE ...
),
C2 AS
(
  SELECT
  FROM  C1
  WHERE ...
)
SELECT ...
FROM C2
WHERE ...;
```

Because the CTE name is assigned before the start of the outer query, you can refer to multiple instances of the same CTE name, unlike with derived tables. The general form looks like the following.

```
WITH C AS
(
  SELECT ...
  FROM T1
)
SELECT ...
FROM C AS C1
  INNER JOIN C AS C2
    ON ...;
```

EXAM TIP

Since in the last example the outer query has multiple reference to the same CTE name, the underlying work is going to be repeated. If the inner query performs expensive work, and generates a fairly small result set, a table expression might not be the best choice. In such a case you should consider persisting the inner query's result in a temporary table first, and then joining two instances of that temporary table.

CTEs also have a recursive form. The body of the recursive query has two or more queries, usually separated by a UNION ALL operator. At least one of the queries in the CTE body, known as the anchor member, is a query that returns a valid relational result. The anchor query is invoked only once. In addition, at least one of the queries in the CTE body, known as the recursive member, has a reference to the CTE name. This query is invoked repeatedly until it returns an empty result set. In each iteration, the reference to the CTE name from the recursive member represents the previous result set. Then the reference to the CTE name from

the outer query represents the unified results of the invocation of the anchor member and all invocations of the recursive member.

As an example, the following code uses a recursive CTE to return the management chain, leading all the way up to the CEO for a specified employee:

```
WITH EmpsCTE AS
(
  SELECT empid, mgrid, firstname, lastname, 0 AS distance
  FROM HR.Employees
  WHERE empid = 9

  UNION ALL

  SELECT M.empid, M.mgrid, M.firstname, M.lastname, S.distance + 1 AS distance
  FROM EmpsCTE AS S
    JOIN HR.Employees AS M
      ON S.mgrid = M.empid
)
SELECT empid, mgrid, firstname, lastname, distance
FROM EmpsCTE;
```

This code returns the following output.

```
empid       mgrid       firstname  lastname             distance
----------- ----------- ---------- -------------------- -----------
9           5           Patricia   Doyle                0
5           2           Sven       Mortensen            1
2           1           Don        Funk                 2
1           NULL        Sara       Davis                3
```

As you can see, the anchor member returns the row for employee 9. Then the recursive member is invoked repeatedly, and in each round joins the previous result set with the HR.Employees table to return the direct manager of the employee from the previous round. The recursive query stops as soon as it returns an empty set—in this case, after not finding a manager of the CEO. Then the outer query returns the unified results of the invocation of the anchor member (the row for employee 9) and all invocations of the recursive member (all managers above employee 9).

Views and inline table-valued functions

As you learned in the previous sections, derived tables and CTEs are table expressions that are visible only in the scope of the statement that defines them. After that statement terminates, the table expression is gone. Hence, derived tables and CTEs are not reusable. For reusability, you need to store the definition of the table expression as an object in the database, and for this you can use either views or inline table-valued functions. Because these are objects in the database, you can control access by assigning permissions.

The main difference between views and inline table-valued functions is that the former doesn't accept input parameters and the latter does. As an example, suppose you need to

persist the definition of the query with the row number computation from the examples in the previous sections. To achieve this, you create the following view:

```
DROP VIEW IF EXISTS Sales.RankedProducts;
GO
CREATE VIEW Sales.RankedProducts
AS

SELECT
  ROW_NUMBER() OVER(PARTITION BY categoryid
                    ORDER BY unitprice, productid) AS rownum,
  categoryid, productid, productname, unitprice
FROM Production.Products;
GO
```

Note that it's not the result set of the view that is stored in the database; rather, only its definition is stored. Now that the definition is stored, the object is reusable. Whenever you need to query the view, it's available to you, assuming you have the permissions to query it:

```
SELECT categoryid, productid, productname, unitprice
FROM Sales.RankedProducts
WHERE rownum <= 2;
```

As for inline table-valued functions, they are very similar to views in concept; however, as mentioned, they do support input parameters. So if you want to define something like a view with parameters, the closest you have is an inline table-valued function. As an example, consider the recursive CTE from the section about CTEs that retuned the management chain leading all the way up to the CEO for a specified employee. Suppose that you wanted to encapsulate the logic in a table expression for reusability, but also wanted to parameterize the input employee instead of using the constant 9. You can achieve this by using an inline table-valued function with the following definition:

```
DROP FUNCTION IF EXISTS HR.GetManagers;
GO
CREATE FUNCTION HR.GetManagers(@empid AS INT) RETURNS TABLE
AS

RETURN
  WITH EmpsCTE AS
  (
    SELECT empid, mgrid, firstname, lastname, 0 AS distance
    FROM HR.Employees
    WHERE empid = @empid

    UNION ALL

    SELECT M.empid, M.mgrid, M.firstname, M.lastname, S.distance + 1 AS distance
    FROM EmpsCTE AS S
      JOIN HR.Employees AS M
        ON S.mgrid = M.empid
  )
  SELECT empid, mgrid, firstname, lastname, distance
  FROM EmpsCTE;
GO
```

Observe that the header assigns the function with a name (HR.GetManagers), defines the input parameter (@empid AS INT), and indicates that the function returns a table result (defined by the returned query). Then the function has a RETURN clause returning the result of the recursive query, and the anchor member of the recursive CTE filters the employee whose ID is equal to the input employee ID. When querying the function, you pass a specific input employee ID as the following example shows:

```
SELECT *
FROM HR.GetManagers(9) AS M;
```

EXAM TIP

You're not limited to only issuing SELECT statements against table expressions, rather you can also modify data through them. The outer statement could be INSERT, UPDATE, DELETE and MERGE. Since the table expression is merely a reflection of data from some underlying table, it's the underlying table that is actually modified.

When you're done, run the following code for cleanup:

```
DROP VIEW IF EXISTS Sales.RankedProducts;
DROP FUNCTION IF EXISTS HR.GetManagers;
```

Skill 2.3: Group and pivot data by using queries

This skill covers various data analysis tools that T-SQL supports. It covers aggregating data with both grouped queries and windowed queries. It covers the ability to group the data in multiple different ways by defining what's called grouping sets. It also covers data rotation with pivoting and unpivoting techniques.

This section covers how to:

- Use windowing functions to group and rank the results of a query
- Distinguish between using windowing functions and GROUP BY
- Construct complex GROUP BY clauses using GROUPING SETS, and CUBE
- Construct PIVOT and UNPIVOT statements to return desired results based on supplied data
- Determine the impact of NULL values in PIVOT and UNPIVOT queries

Writing grouped queries

You can use grouped queries to define groups in your data, and then you can perform data analysis computations per group. You group the data by a set of expressions known as a grouping set. Traditional T-SQL queries define a single grouping set; namely, they group the data in only one way. T-SQL also supports defining multiple grouping sets in one query.

Working with a single grouping set

With grouped queries, you can arrange the rows you're querying in groups and apply data analysis computations like aggregate functions against those groups. A query becomes a grouped query when you use a group function, a GROUP BY clause, or both.

A query that invokes a group aggregate function but doesn't have an explicit GROUP BY clause arranges all rows in one group. Such an aggregate is referred to as a *scalar aggregate*. Consider the following query as an example:

```
USE TSQLV4;

SELECT COUNT(*) AS numorders
FROM Sales.Orders;
```

This query generates the following output.

```
numorders
-----------
830
```

Because there's no explicit GROUP BY clause, all rows queried from the Sales.Orders table are arranged in one group, and then the COUNT(*) function counts the number of rows in that group. Grouped queries return one result row per group, and because the query defines only one group, it returns only one row in the result set.

Using an explicit GROUP BY clause, you can group the rows based on a specified grouping set of expressions. For example, the following query groups the rows by shipper ID and counts the number of rows (orders, in this case) per distinct group:

```
SELECT shipperid, COUNT(*) AS numorders
FROM Sales.Orders
GROUP BY shipperid;
```

This query generates the following output:

```
shipperid   numorders
----------- -----------
1           249
2           326
3           255
```

The query identifies three groups because there are three distinct shipper IDs.

The grouping set can be made of multiple elements. For example, the following query groups the rows by shipper ID and shipped year:

```
SELECT shipperid, YEAR(shippeddate) AS shippedyear,
   COUNT(*) AS numorders
FROM Sales.Orders
GROUP BY shipperid, YEAR(shippeddate);
```

This query generates the following output:

```
shipperid   shippedyear numorders
----------- ----------- -----------
1           2014        36
2           2015        143
1           NULL        4
3           2016        73
3           NULL        6
3           2014        51
2           2016        116
2           NULL        11
1           2015        130
3           2015        125
1           2016        79
2           2014        56
```

Notice that you get a group for each distinct shipper ID and shipped year combination that exists in the data, even when the shipped year is NULL. Remember that a NULL in the `shippeddate` column represents unshipped orders, so a NULL in the `shippedyear` column represents the group of unshipped orders for the respective shipper.

If you need to filter entire groups, you need a filtering option that is evaluated at the group level—unlike the WHERE clause, which is evaluated at the row level. For this, T-SQL provides the HAVING clause. Like the WHERE clause, the HAVING clause uses a predicate but evaluates the predicate per group as opposed to per row. This means that you can refer to aggregate computations because the data has already been grouped.

For example, suppose that you need to group only shipped orders by shipper ID and shipping year, and filter only groups having fewer than 100 orders. You can use the following query to achieve this task:

```
SELECT shipperid, YEAR(shippeddate) AS shippedyear,
   COUNT(*) AS numorders
FROM Sales.Orders
WHERE shippeddate IS NOT NULL
GROUP BY shipperid, YEAR(shippeddate)
HAVING COUNT(*) < 100;
```

This query generates the following output.

```
shipperid   shippedyear numorders
----------- ----------- -----------
1           2014        36
3           2016        73
3           2014        51
1           2016        79
2           2014        56
```

Notice that the query filters only shipped orders in the WHERE clause. This filter is applied at the row level conceptually before the data is grouped. Next the query groups the data by shipper ID and shipped year. Then the HAVING clause filters only groups that have a count of rows (orders) that is less than 100. Finally, the SELECT clause returns the shipper ID, shipped year, and count of orders per remaining group.

T-SQL supports a number of aggregate functions. Those include COUNT(*) and a few general set functions (as they are categorized by standard SQL) like COUNT, SUM, AVG, MIN, and MAX. General set functions are applied to an expression and ignore NULLs.

The following query invokes the COUNT(*) function, in addition to a number of general set functions, including COUNT:

```
SELECT shipperid,
  COUNT(*) AS numorders,
  COUNT(shippeddate) AS shippedorders,
  MIN(shippeddate) AS firstshipdate,
  MAX(shippeddate) AS lastshipdate,
  SUM(val) AS totalvalue
FROM Sales.OrderValues
GROUP BY shipperid;
```

This query generates the following output:

```
shipperid   numorders   shippedorders firstshipdate lastshipdate totalvalue
----------- ----------- ------------- ------------- ------------ -----------
3           255         249           2014-07-15    2016-05-01   383405.53
1           249         245           2014-07-10    2016-05-04   348840.00
2           326         315           2014-07-11    2016-05-06   533547.69
```

Notice the difference between the results of COUNT(shippeddate) and COUNT(*). The former ignores NULLs in the `shippeddate` column, and therefore the counts are less than or equal to those produced by the latter.

With aggregate functions, you can work with distinct occurrences by specifying a DISTINCT clause before the expression, as follows:

```
SELECT shipperid, COUNT(DISTINCT shippeddate) AS numshippingdates
FROM Sales.Orders
GROUP BY shipperid;
```

This query generates the following output.

```
shipperid   numshippingdates
----------- -----------------
1           188
2           215
3           198
```

Note that the DISTINCT option is available not only to the COUNT function, but also to other general set functions. However, it's more common to use it with COUNT.

From a logical query processing perspective, the GROUP BY clause is evaluated after the FROM and WHERE clauses, and before the HAVING, SELECT, and ORDER BY clauses. So the last three clauses already work with a grouped table, and therefore the expressions that they support are limited. Each group is represented by only one result row; therefore, all expressions that appear in those clauses must guarantee a single result value per group. There's no problem referring directly to elements that appear in the GROUP BY clause because each of those returns only one distinct value per group. But if you want to refer to elements from the underlying tables that don't appear in the GROUP BY list, you must apply an aggregate function to them. That's how you can be sure that the expression returns only one value per group. As an example, the following query isn't valid:

```
SELECT S.shipperid, S.companyname, COUNT(*) AS numorders
FROM Sales.Shippers AS S
  INNER JOIN Sales.Orders AS O
    ON S.shipperid = O.shipperid
GROUP BY S.shipperid;
```

This query generates the following error:

```
Msg 8120, Level 16, State 1, Line 58
Column 'Sales.Shippers.companyname' is invalid in the select list because it is not
contained in either an aggregate function or the GROUP BY clause.
```

Even though you know that there can't be more than one distinct company name per distinct shipper ID, T-SQL doesn't know this. Because the S.companyname column neither appears in the GROUP BY list nor is it contained in an aggregate function, it's not allowed in the HAVING, SELECT, and ORDER BY clauses.

You can use a number of workarounds. One solution is to add the S.companyname column to the GROUP BY list, as follows:

```
SELECT S.shipperid, S.companyname,
  COUNT(*) AS numorders
FROM Sales.Shippers AS S
  INNER JOIN Sales.Orders AS O
    ON S.shipperid = O.shipperid
GROUP BY S.shipperid, S.companyname;
```

This query generates the following output.

```
shipperid   companyname    numorders
----------- -------------- -----------
1           Shipper GVSUA  249
2           Shipper ETYNR  326
3           Shipper ZHISN  255
```

Another workaround is to apply an aggregate function like MAX to the column, as follows:

```
SELECT S.shipperid,
  MAX(S.companyname) AS companyname,
  COUNT(*) AS numorders
FROM Sales.Shippers AS S
  INNER JOIN Sales.Orders AS O
    ON S.shipperid = O.shipperid
GROUP BY S.shipperid;
```

In this case, the aggregate function is an artificial one because there can't be more than one distinct company name per distinct shipper ID. The first workaround, though, tends to produce more optimal plans, and also seems to be the more natural solution.

The third workaround is to group and aggregate the rows from the Orders table first, define a table expression based on the grouped query, and then join the table expression with the Shippers table to get the shipper company names. Here's the solution's code:

```
WITH C AS
(
  SELECT shipperid, COUNT(*) AS numorders
  FROM Sales.Orders
  GROUP BY shipperid
)
SELECT S.shipperid, S.companyname, numorders
FROM Sales.Shippers AS S
  INNER JOIN C
    ON S.shipperid = C.shipperid;
```

SQL Server usually optimizes the third solution like it does the first. The first solution might be preferable because it involves much less code.

> ***MORE INFO*** **ON USER DEFINED AGGREGATES**
>
> **SQL Server also allows you to create user defined aggregates (UDA) using .NET code based on the Common Language Runtime (CLR). It provides some built-in CLR UDAs for the spatial data types GEOMETRY and GEOGRAPHY and also allows you to create new UDAs operating on spatial types as inputs. For more details see the topic "CLR User-Defined Aggregates" in books online at *https://msdn.microsoft.com/en-us/library/ms131057.aspx*.**

Working with multiple grouping sets

With T-SQL, you can define multiple grouping sets in the same query. In other words, you can use one query to group the data in more than one way. T-SQL supports three clauses that allow defined multiple grouping sets: GROUPING SETS, CUBE, and ROLLUP. You use these in the GROUP BY clause.

You can use the GROUPING SETS clause to list all grouping sets that you want to define in the query. As an example, the following query defines four grouping sets:

```
SELECT shipperid, YEAR(shippeddate) AS shipyear, COUNT(*) AS numorders
FROM Sales.Orders
WHERE shippeddate IS NOT NULL -- exclude unshipped orders
GROUP BY GROUPING SETS
(
  ( shipperid, YEAR(shippeddate) ),
  ( shipperid                    ),
  ( YEAR(shippeddate)            ),
  (                              )
);
```

You list the grouping sets separated by commas within the outer pair of parentheses, which belongs to the GROUPING SETS clause. You use an inner pair of parentheses to enclose each grouping set. If you don't indicate an inner pair of parentheses, each individual element is considered a separate grouping set.

This query defines four grouping sets. One of them is the empty grouping set, which defines one group with all rows for computation of grand aggregates. The query generates the following output:

```
shipperid   shipyear    numorders
----------- ----------- -----------
1           2014        36
2           2014        56
3           2014        51
NULL        2014        143
1           2015        130
2           2015        143
3           2015        125
NULL        2015        398
1           2016        79
2           2016        116
3           2016        73
NULL        2016        268
NULL        NULL        809
3           NULL        249
1           NULL        245
2           NULL        315
```

The output combines the results of grouping and aggregating the data of four different grouping sets. As you can see in the output, NULLs are used as placeholders in rows where an element isn't part of the grouping set. For example, in result rows that are associated with the

grouping set (shipperid), the `shipyear` result column is set to NULL. Similarly, in rows that are associated with the grouping set (YEAR(shippeddate)), the `shipperid` column is set to NULL.

You can achieve the same result by writing four separate grouped queries—each defining only a single grouping set—and unifying their results with a UNION ALL operator. However, such a solution would involve much more code and won't get optimized as efficiently as the query with the GROUPING SETS clause.

T-SQL supports two additional clauses called CUBE and ROLLUP, which you can consider as abbreviations of the GROUPING SETS clause. The CUBE clause accepts a list of expressions as inputs and defines all possible grouping sets that can be generated from the inputs—including the empty grouping set. For example, the following query is a logical equivalent of the previous query that used the GROUPING SETS clause:

```
SELECT shipperid, YEAR(shippeddate) AS shipyear, COUNT(*) AS numorders
FROM Sales.Orders
WHERE shippeddate IS NOT NULL
GROUP BY CUBE( shipperid, YEAR(shippeddate) );
```

The CUBE clause defines all four possible grouping sets from the two inputs:

- (shipperid, YEAR(shippeddate))
- (shipperid)
- (YEAR(shippeddate))
- ( )

The ROLLUP clause is also an abbreviation of the GROUPING SETS clause, but you use it when there's a natural hierarchy formed by the input elements. In such a case, only a subset of the possible grouping sets is really interesting. Consider, for example, a location hierarchy made of the elements `shipcountry`, `shipregion`, and `shipcity`, in this order. It's only interesting to roll up the data in one direction, computing aggregates for the following grouping sets:

- (shipcountry, shipregion, shipcity)
- (shipcountry, shipregion)
- (shipcountry)
- ( )

The other grouping sets are simply not interesting. For example, even though the same city name can appear in different places in the world, it's not interesting to aggregate all of the occurrences—irrespective of region and country.

So, when the elements form a hierarchy, you use the ROLLUP clause and this way avoid computing unnecessary aggregates. Here's an example of a query using the ROLLUP clause based on the aforementioned hierarchy:

```
SELECT shipcountry, shipregion, shipcity, COUNT(*) AS numorders
FROM Sales.Orders
GROUP BY ROLLUP( shipcountry, shipregion, shipcity );
```

This query generates the following output (shown here in abbreviated form):

```
shipcountry     shipregion      shipcity        numorders
--------------- --------------- --------------- -----------
Argentina       NULL            Buenos Aires    16
Argentina       NULL            NULL            16
Argentina       NULL            NULL            16
...
USA             AK              Anchorage       10
USA             AK              NULL            10
USA             CA              San Francisco   4
USA             CA              NULL            4
USA             ID              Boise           31
USA             ID              NULL            31
...
USA             NULL            NULL            122
...
NULL            NULL            NULL            830
```

As mentioned, NULLs are used as placeholders when an element isn't part of the grouping set. If all grouped columns disallow NULLs in the underlying table, you can identify the rows that are associated with a single grouping set based on a unique combination of NULLs and non-NULLs in those columns. A problem arises in identifying the rows that are associated with a single grouping set when a grouped column allows NULLs—as is the case with the `shipregion` column. How do you tell whether a NULL in the result represents a placeholder (meaning "all regions") or an original NULL from the table (meaning "missing region")? T-SQL provides two functions to help address this problem: GROUPING and GROUPING_ID.

The GROUPING function accepts a single element as input and returns 0 when the element is part of the grouping set and 1 when it isn't. In other words, 0 defines a detail element and 1 defines a hyperaggregate. The following query demonstrates using the GROUPING function:

```
SELECT
  shipcountry, GROUPING(shipcountry) AS grpcountry,
  shipregion , GROUPING(shipregion)   AS grpregion,
  shipcity   , GROUPING(shipcity)    AS grpcity,
  COUNT(*) AS numorders
FROM Sales.Orders
GROUP BY ROLLUP( shipcountry, shipregion, shipcity );
```

This query generates the following output (shown here in abbreviated form):

```
shipcountry   grpcountry  shipregion   grpregion   shipcity      grpcitry    numorders
------------  ----------  -----------  ----------  ------------- ----------  ----------
Argentina     0           NULL         0           Buenos Aires   0           16
Argentina     0           NULL         0           NULL           1           16
Argentina     0           NULL         1           NULL           1           16
...
USA           0           AK           0           Anchorage      0           10
USA           0           AK           0           NULL           1           10
USA           0           CA           0           San Francisco  0           4
```

```
USA             0           CA              0           NULL            1           4
USA             0           ID              0           Boise           0           31
USA             0           ID              0           NULL            1           31
...
USA             0           NULL            1           NULL            1           122
...
NULL            1           NULL            1           NULL            1           830
```

Now you can identify a grouping set by looking for 0s in the elements that are part of the grouping set (detail elements) and 1s in the rest (aggregate elements).

Another function that you can use to identify the grouping sets is GROUPING_ID. This function accepts the list of grouped columns as inputs and returns an integer representing a bitmap. The rightmost bit represents the rightmost input. The bit is 0 when the respective element is part of the grouping set and 1 when it isn't. Each bit represents 2 raised to the power of the bit position minus 1; so, the rightmost bit represents 1, the one to the left of it 2, then 4, then 8, and so on. The result integer is the sum of the values representing elements that are not part of the grouping set because their bits are turned on. Here's a query demonstrating the use of this function:

```
SELECT GROUPING_ID( shipcountry, shipregion, shipcity ) AS grp_id,
  shipcountry, shipregion, shipcity,
  COUNT(*) AS numorders
FROM Sales.Orders
GROUP BY ROLLUP( shipcountry, shipregion, shipcity );
```

This query generates the following output (shown here in abbreviated form):

```
grp_id      shipcountry     shipregion      shipcity        numorders
----------- --------------- --------------- --------------- -----------
0           Argentina       NULL            Buenos Aires    16
1           Argentina       NULL            NULL            16
3           Argentina       NULL            NULL            16
...
0           USA             AK              Anchorage        10
1           USA             AK              NULL            10
0           USA             CA              San Francisco   4
1           USA             CA              NULL            4
0           USA             ID              Boise           31
1           USA             ID              NULL            31
...
3           USA             NULL            NULL            122
...
7           NULL            NULL            NULL            830
```

The last row in this output represents the empty grouping set—none of the three elements is part of the grouping set. Therefore, the respective bits (values 1, 2, and 4) are turned on. The sum of the values that those bits represent is 7.

NOTE GROUPING SETS ALGEBRA

You can specify multiple GROUPING SETS, CUBE, and ROLLUP clauses in the GROUP BY clause separated by commas. By doing so, you achieve a multiplication effect. For example the clause CUBE(a, b, c) defines eight grouping sets and the clause ROLLUP(x, y, z) defines four grouping sets. By specifying a comma between the two, as in CUBE(a, b, c), ROLLUP(x, y, z), you multiply them and get 32 grouping sets. If you place a CUBE or ROLLUP clause within a GROUPING SETS clause, you achieve an addition effect. For example, the expression GROUPING SETS((x, y, z), (z), CUBE(a, b, c)) adds the two grouping sets defined explicitly by the GROUPING SETS clause and the eight defined implicitly by the CUBE clause and produces ten grouping sets in total.

MORE INFO ON GROUPED QUERIES

For coverage of the logical query processing aspects of grouping, see "Logical Query Processing Part 7: GROUP BY and HAVING" at *http://sqlmag.com/sql-server/logical-query-processing-part-7-group-and-having*.

Pivoting and Unpivoting Data

Pivoting is a specialized case of grouping and aggregating of data. Unpivoting is, in a sense, the inverse of pivoting. T-SQL supports native operators for both. Let's first describe the PIVOT operator and then the UNPIVOT operator.

Pivoting Data

Pivoting is a technique that groups and aggregates data, transitioning it from a state of rows to a state of columns. In all pivot queries, you need to identify three elements:

1. What do you want to see on rows? This element is known as the *on rows*, or *grouping* element.
2. What do you want to see on columns? This element is known as the *on cols*, or *spreading* element.
3. What do you want to see in the intersection of each distinct row and column value? This element is known as the *data*, or *aggregation* element.

As an example of a pivot request, suppose that you want to query the Sales.Orders table. You want to return a row for each distinct customer ID (the grouping element), a column for each distinct shipper ID (the spreading element), and in the intersection of each customer and shipper you want to see the sum of freight values (the aggregation element). With T-SQL, you can achieve such a pivoting task by using the PIVOT table operator. The recommended form for a pivot query (more on why it's the recommended form later) is generally like the following.

```
WITH PivotData AS
(
  SELECT
    < grouping column >,
    < spreading column >,
    < aggregation column >
  FROM < source table >
)
SELECT < select list >
FROM PivotData
  PIVOT( < aggregate function >(< aggregation column >)
    FOR < spreading column > IN (< distinct spreading values >) ) AS P;
```

This recommended general form is made of the following elements:

- You define a table expression (like the one named PivotData) that returns the three elements that are involved in pivoting, which in this example are `custid`, `shipperid` and `freight` from Sales.Orders. It is not recommended to query the underlying source table directly; the reason for this is explained shortly.
- You issue the outer query against the table expression and apply the PIVOT operator to that table expression. The PIVOT operator returns a table result. You need to assign an alias to that table, for example, P.
- The specification for the PIVOT operator starts by indicating an aggregate function applied to the aggregation element—in this example, SUM(freight).
- Then you specify the FOR clause followed by the spreading column, which in this example is `shipperid`.
- Then you specify the IN clause followed by the list of distinct values that appear in the spreading element, separated by commas. What used to be values in the spreading column (in this example, shipper IDs) become column names in the result table. Therefore, the items in the list should be expressed as column identifiers. Remember that if a column identifier is irregular, it has to be delimited. Because shipper IDs are integers, they have to be delimited: [1],[2],[3].

Following this recommended syntax for pivot queries, the following query addresses our task (return customer IDs on rows, shipper IDs on columns, and the total freight in the intersections):

```
WITH PivotData AS
(
  SELECT
    custid,    -- grouping column
    shipperid, -- spreading column
    freight    -- aggregation column
  FROM Sales.Orders
)
SELECT custid, [1], [2], [3]
FROM PivotData
  PIVOT(SUM(freight) FOR shipperid IN ([1],[2],[3]) ) AS P;
```

This query generates the following output (shown here in abbreviated form):

```
custid  1        2         3
------- -------- --------- --------
1       95.03    61.02     69.53
2       43.90    NULL      53.52
3       63.09    116.56    88.87
4       41.95    358.54    71.46
5       189.44   1074.51   295.57
6       0.15     126.19    41.92
7       217.96   215.70    190.00
8       16.16    175.01    NULL
9       341.16   419.57    597.14
10      129.42   162.17    502.36
...

(89 row(s) affected)
```

EXAM TIP

As you can see in the output, in cases where a shipper hasn't shipped any orders for a customer, the result of the aggregate is NULL. If you need to return something else instead of a NULL in such cases, say, zero, apply the ISNULL or COALESCE function to the result columns in the outer query's SELECT list. For example ISNULL([1], 0.00). Your revised query would look like this:

```
WITH PivotData AS
(
  SELECT
    custid,
    shipperid,
    freight
  FROM Sales.Orders
)
SELECT custid,
  ISNULL([1], 0.00) AS [1],
  ISNULL([2], 0.00) AS [2],
  ISNULL([3], 0.00) AS [3]
FROM PivotData
  PIVOT(SUM(freight) FOR shipperid IN ([1],[2],[3]) ) AS P;
```

This code generates the following output:

```
custid  1        2         3
------- -------- --------- --------
1       95.03    61.02     69.53
2       43.90    0.00      53.52
3       63.09    116.56    88.87
4       41.95    358.54    71.46
5       189.44   1074.51   295.57
6       0.15     126.19    41.92
7       217.96   215.70    190.00
8       16.16    175.01    0.00
9       341.16   419.57    597.14
10      129.42   162.17    502.36
...
```

Make sure that whenever you write T-SQL code you always think about NULLs and how you want to handle them.

If you look carefully at the specification of the PIVOT operator, you will notice that you indicate the aggregation and spreading elements, but not the grouping element. The grouping element is identified by elimination—it's what's left from the queried table besides the aggregation and spreading elements. This is why it is recommended to prepare a table expression for the pivot operator returning only the three elements that should be involved in the pivoting task. If you query the underlying table directly (Sales.Orders in this case), all columns from the table besides the aggregation (freight) and spreading (shipperid) columns will implicitly become your grouping elements. This includes even the primary key column orderid. So instead of getting a row per customer, you end up getting a row per order. You can see it for yourself by running the following code:

```
SELECT custid, [1], [2], [3]
FROM Sales.Orders
  PIVOT(SUM(freight) FOR shipperid IN ([1],[2],[3]) ) AS P;
```

This query generates the following output (shown here in abbreviated form):

```
custid  1       2       3
------- ------- ------- -------
85      NULL    NULL    32.38
79      11.61   NULL    NULL
34      NULL    65.83   NULL
84      41.34   NULL    NULL
76      NULL    51.30   NULL
34      NULL    58.17   NULL
14      NULL    22.98   NULL
68      NULL    NULL    148.33
88      NULL    13.97   NULL
35      NULL    NULL    81.91
...

(830 row(s) affected)
```

You get 830 rows back because there are 830 rows in the Sales.Orders table. By defining a table expression as was shown in the recommended solution, you control which columns will be used as the grouping columns. If you return `custid`, `shipperid`, and `freight` in the table expression, and use the last two as the spreading and aggregation elements, respectively, the PIVOT operator implicitly assumes that `custid` is the grouping element. Therefore, it groups the data by `custid`, and as a result, returns a single row per customer.

You should be aware of a few limitations of the PIVOT operator.

- The aggregation and spreading elements cannot directly be results of expressions; instead, they must be column names from the queried table. You can, however, apply expressions in the query defining the table expression, assign aliases to those expressions, and then use the aliases in the PIVOT operator.
- The COUNT(*) function isn't allowed as the aggregate function used by the PIVOT operator. If you need a count, you have to use the general COUNT(<col name>) aggregate function. A simple workaround is to define a dummy column in the table expression made of a constant, as in 1 AS agg_col, and then in the PIVOT operator apply the aggregate function to that column: COUNT(agg_col). In this case you can also use SUM(agg_col) as an alternative.
- A PIVOT operator is limited to using only one aggregate function.
- The IN clause of the PIVOT operator accepts a static list of spreading values. It doesn't support a subquery as input. You need to know ahead what the distinct values are in the spreading column and specify those in the IN clause. When the list isn't known ahead, you can use dynamic SQL to construct and execute the query string after querying the distinct values from the data. You can find an example for building and executing a pivot query dynamically at *http://sqlmag.com/sql-server/logical-query-processing-clause-and-pivot*.

Unpivoting Data

Unpivoting data can be considered the inverse of pivoting. The starting point is some pivoted data. When unpivoting data, you rotate the input data from a state of columns to a state of rows. Just like T-SQL supports the native PIVOT table operator to perform pivoting, it supports a native UNPIVOT operator to perform unpivoting. Like PIVOT, UNPIVOT is implemented as a table operator that you use in the FROM clause. The operator operates on the input table that is provided to its left, which could be the result of other table operators, like joins. The outcome of the UNPIVOT operator is a table result that can be used as the input to other table operators that appear to its right.

To demonstrate unpivoting, use as an example a sample table called Sales.FreightTotals. The following code creates the sample data and queries it to show its contents:

```
USE TSQLV4;
DROP IF EXISTS TABLE Sales.FreightTotals;
GO

WITH PivotData AS
(
  SELECT
    custid,    -- grouping column
    shipperid, -- spreading column
    freight    -- aggregation column
  FROM Sales.Orders
)
SELECT *
INTO Sales.FreightTotals
```

```
FROM PivotData
  PIVOT( SUM(freight) FOR shipperid IN ([1],[2],[3]) ) AS P;

SELECT * FROM Sales.FreightTotals;
```

This code generates the following output, shown here in abbreviated form:

```
custid  1        2        3
------- -------- -------- --------
1       95.03    61.02    69.53
2       43.90    NULL     53.52
3       63.09    116.56   88.87
4       41.95    358.54   71.46
5       189.44   1074.51  295.57
6       0.15     126.19   41.92
7       217.96   215.70   190.00
8       16.16    175.01   NULL
9       341.16   419.57   597.14
10       129.42   162.17   502.36
...
```

As you can see, the source table has a row for each customer and a column for each shipper (shippers 1, 2, and 3). The intersection of each customer and shipper has the total freight values. The unpivoting task at hand is to return a row for each customer and shipper holding the customer ID in one column, the shipper ID in a second column, and the freight value in a third column.

Unpivoting always takes a set of source columns and rotates those to multiple rows, generating two target columns: one to hold the source column values and another to hold the source column names. The source columns already exist, so their names should be known to you. But the two target columns are created by the unpivoting solution, so you need to choose names for those. In our example, the source columns are [1], [2], and [3]. As for names for the target columns, you need to decide on those. In this case, it might be suitable to call the values column freight and the names column `shipperid`. So remember, in every unpivoting task, you need to identify the three elements involved:

1. The name you want to assign to the target values column (in this case, freight).
2. The name you want to assign to the target names column (in this case, shipperid).
3. The set of source columns that you're unpivoting (in this case, [1],[2],[3]).

After you identify these three elements, you use the following query form to handle the unpivoting task:

```
SELECT < column list >, < names column >, < values column >
FROM < source table >
  UNPIVOT( < values column > FOR < names column > IN( <source columns> ) ) AS U;
```

Based on this syntax, the following query addresses the current task:

```
SELECT custid, shipperid, freight
FROM Sales.FreightTtotals
  UNPIVOT( freight FOR shipperid IN([1],[2],[3]) ) AS U;
```

This query generates the following output (shown here in abbreviated form).

```
custid  shipperid  freight
------- ---------- --------
1       1          95.03
1       2          61.02
1       3          69.53
2       1          43.90
2       3          53.52
3       1          63.09
3       2          116.56
3       3          88.87
4       1          41.95
4       2          358.54
4       3          71.46
...
```

NOTE **UNPIVOT AND NULLS**

Besides unpivoting the data, the UNPIVOT operator filters out rows with NULLs in the value column (freight in this case). The assumption is that those represent inapplicable cases. There was no escape from keeping NULLs in the source if the column was applicable to at least one other customer. But after unpivoting the data, there's no reason to keep a row for a certain customer-shipper pair if it's inapplicable—if that shipper did not ship orders to that customer. However, if you want to return rows for cases that were NULL originally, you need to prepare a table expression where you replace the NULLs with some other value using the ISNULL or COALESCE function, and then in the outer query replace that value back with a NULL using the NULLIF function. Here's how you apply this technique to our example:

```
WITH C AS
(
  SELECT custid,
    ISNULL([1], 0.00) AS [1],
    ISNULL([2], 0.00) AS [2],
    ISNULL([3], 0.00) AS [3]
  FROM Sales.FreightTotals
)
SELECT custid, shipperid, NULLIF(freight, 0.00) AS freight
FROM C
  UNPIVOT( freight FOR shipperid IN([1],[2],[3]) ) AS U;
```

This time the output includes rows with NULLs:

```
custid  shipperid  freight
------- ---------- --------
1       1          95.03
1       2          61.02
1       3          69.53
2       1          43.90
2       2          NULL
2       3          53.52
3       1          63.09
3       2          116.56
```

```
3       3           88.87
4       1           41.95
4       2           358.54
4       3           71.46
...
```

Naturally you need to replace the NULL with a value that normally can't appear in your data.

In terms of data types, the names column is defined as a Unicode character string (NVARCHAR(128)). The values column is defined with the same type as the type of the source columns that were unpivoted. For this reason, the types of all columns that you're unpivoting must be the same.

Just like with the PIVOT operator, in a static query the UNPIVOT operator requires you to hard code the columns that you want to unpivot. If you don't want to hard code those, you can query the column names from the sys.columns view, construct the UNPIVOT query string, and execute it with dynamic SQL. Also, the UNPIVOT operator is limited to unpivoting only one measure (one values column). If you need to unpivot multiple measures, you need to use an alternative solution that is based on the APPLY operator. The below resources provide further reading material with examples for constructing and executing dynamic queries as well as unpivoting multiple measures.

When you're done, run the following code for cleanup:

```
DROP TABLE IF EXISTS Sales.FreightTotals;
```

> ***MORE INFO*** **ON PIVOT AND UNPIVOT**
>
> **For more information about the logical query processing aspects of the PIVOT and UNPIVOT operators, see "Logical Query Processing: The FROM Clause and PIVOT" at *https://sqlmag.com/sql-server/logical-query-processing-clause-and-pivot* and "Logical Query Processing Part 5: The FROM Clause and UNPIVOT" at *https://sqlmag.com/sql-server/logical-query-processing-part-5-clause-and-unpivot*.**

Using Window Functions

Like group functions, window functions also enable you to perform data analysis computations. The difference between the two is in how you define the set of rows for the function to work with. With group functions, you use grouped queries to arrange the queried rows in groups, and then the group functions are applied to each group. You get one result row per group—not per underlying row. With window functions, you define the set of rows per function—and then return one result value per underlying row and function. You define the set of rows for the function to work with using a clause called OVER.

This section covers three types of window functions: aggregate, ranking, and offset.

Window aggregate functions

Window aggregate functions are the same as the group aggregate functions (for example, SUM, COUNT, AVG, MIN, and MAX), except window aggregate functions are applied to a window of rows defined by the OVER clause.

One of the benefits of using window functions is that unlike grouped queries, windowed queries do not hide the detail—they return a row for every underlying query's row. This means that you can mix detail and aggregated elements in the same query, and even in the same expression. Using the OVER clause, you define a set of rows for the function to work with per underlying row. In other words, a windowed query defines a window of rows per function and row in the underlying query.

As mentioned, you use an OVER clause to define a window of rows for the function. The window is defined with respect to the current row. When using empty parentheses, the OVER clause represents the entire underlying query's result set. For example, the expression SUM(val) OVER() represents the grand total of all rows in the underlying query. You can use a window partition clause to restrict the window. For example, the expression SUM(val) OVER(PARTITION BY custid) represents the current customer's total. As an example, if the current row has customer ID 1, the OVER clause filters only those rows from the underlying query's result set where the customer ID is 1; hence, the expression returns the total for customer 1.

Here's an example of a query against the Sales.OrderValues view returning for each order the customer ID, order ID, and order value; using window functions, the query also returns the grand total of all values and the customer total:

```
SELECT custid, orderid, val,
  SUM(val) OVER(PARTITION BY custid) AS custtotal,
  SUM(val) OVER() AS grandtotal
FROM Sales.OrderValues;
```

This query generates the following output (shown here in abbreviated form):

```
custid  orderid  val     custtotal  grandtotal
------- -------- ------- ---------- -----------
1       10643    814.50  4273.00    1265793.22
1       10692    878.00  4273.00    1265793.22
1       10702    330.00  4273.00    1265793.22
1       10835    845.80  4273.00    1265793.22
1       10952    471.20  4273.00    1265793.22
1       11011    933.50  4273.00    1265793.22
2       10926    514.40  1402.95    1265793.22
2       10759    320.00  1402.95    1265793.22
2       10625    479.75  1402.95    1265793.22
2       10308    88.80   1402.95    1265793.22
...
```

The grand total is of course the same for all rows. The customer total is the same for all rows with the same customer ID.

You can mix detail elements and windowed aggregates in the same expression. For example, the following query computes for each order the percent of the current order value out of the customer total, and also the percent of the grand total:

```
SELECT custid, orderid, val,
  CAST(100.0 * val / SUM(val) OVER(PARTITION BY custid) AS NUMERIC(5, 2)) AS pctcust,
  CAST(100.0 * val / SUM(val) OVER()                    AS NUMERIC(5, 2)) AS pcttotal
FROM Sales.OrderValues;
```

This query generates the following output (shown here in abbreviated form):

```
custid  orderid  val     pctcust  pcttotal
------- -------- ------- -------- ---------
1       10643    814.50  19.06    0.06
1       10692    878.00  20.55    0.07
1       10702    330.00  7.72     0.03
1       10835    845.80  19.79    0.07
1       10952    471.20  11.03    0.04
1       11011    933.50  21.85    0.07
2       10926    514.40  36.67    0.04
2       10759    320.00  22.81    0.03
2       10625    479.75  34.20    0.04
2       10308    88.80   6.33     0.01
...
```

The sum of all percentages out of the grand total is 100. The sum of all percentages out of the customer total is 100 for each partition of rows with the same customer.

Window aggregate functions support another filtering option called *window frame*. The idea is that you define ordering within the partition by using a window order clause, and then based on that order, you can confine a frame of rows between two delimiters. You define the delimiters by using a window frame clause. The window frame clause requires a window order clause to be present because a set has no order, and without order, limiting rows between two delimiters would have no meaning.

In the window frame clause, you indicate the *window frame unit* (ROWS or RANGE) and the *window frame extent* (the delimiters). With the ROWS window frame unit, you can indicate the delimiters as one of three options:

- UNBOUNDED PRECEDING or FOLLOWING, meaning the beginning or end of the partition, respectively.
- CURRENT ROW, obviously representing the current row.
- <n> ROWS PRECEDING or FOLLOWING, meaning n rows before or after the current, respectively.

As an example, suppose that you wanted to query the Sales.OrderValues view and compute the running total values from the beginning of the current customer's activity until the current order, assuming a sort based on orderdate and orderid as a tiebreaker. You need to use the SUM aggregate. You partition the window by `custid`. You order the window by `orderdate, orderid`. You then frame the rows from the beginning of the partition (UNBOUNDED PRECEDING) until the current row. Your query should look like the following.

```
SELECT custid, orderid, orderdate, val,
  SUM(val) OVER(PARTITION BY custid
                ORDER BY orderdate, orderid
                ROWS BETWEEN UNBOUNDED PRECEDING
                         AND CURRENT ROW) AS runningtotal
FROM Sales.OrderValues;
```

This query generates the following output (shown here in abbreviated form):

```
custid      orderid     orderdate  val     runningtotal
----------- ----------- ---------- ------- -------------
1           10643       2015-08-25 814.50  814.50
1           10692       2015-10-03 878.00  1692.50
1           10702       2015-10-13 330.00  2022.50
1           10835       2016-01-15 845.80  2868.30
1           10952       2016-03-16 471.20  3339.50
1           11011       2016-04-09 933.50  4273.00
2           10308       2014-09-18 88.80   88.80
2           10625       2015-08-08 479.75  568.55
2           10759       2015-11-28 320.00  888.55
2           10926       2016-03-04 514.40  1402.95
...
```

Observe how the values keep accumulating from the beginning of the customer partition until the current row. By the way, instead of the verbose form of the frame extent ROWS BETWEEN UNBOUNDED PRECEDING AND CURRENT ROW, you can use the shorter form ROWS UNBOUNDED PRECEDING, and retain the same meaning.

Using window aggregate functions to perform computations such as running totals, you typically get much better performance compared to using joins or subqueries and group aggregate functions. Window functions lend themselves to good optimization—especially when using UNBOUNDED PRECEDING as the first delimiter.

From an indexing standpoint, an optimal index to support window functions is one created on the partitioning and ordering elements as the key list, and includes the rest of the elements from the query for coverage. I like to think of this index as a *POC index* as an acronym for partitioning, ordering and covering. With such an index in place SQL Server won't need to explicitly sort the data, rather pull it preordered from the index.

EXAM TIP

In terms of logical query processing, a query's result is established when you get to the SELECT phase—after the FROM, WHERE, GROUP BY, and HAVING phases have been processed. Because window functions are supposed to operate on the underlying query's result set, they are allowed only in the SELECT and ORDER BY clauses. If you need to refer to the result of a window function in any clause that is evaluated before the SELECT clause, you need to use a table expression. You invoke the window function in the SELECT clause of the inner query, assigning the expression with a column alias. Then you can refer to that column alias in the outer query in all clauses.

Suppose that you need to filter the result of the last query, returning only those rows where the running total is less than 1,000.00. The following code achieves this by defining a common table expression (CTE) based on the previous query and then doing the filtering in the outer query:

```
WITH RunningTotals AS
(
```

```
  SELECT custid, orderid, orderdate, val,
    SUM(val) OVER(PARTITION BY custid
                  ORDER BY orderdate, orderid
                   ROWS BETWEEN UNBOUNDED PRECEDING
                            AND CURRENT ROW) AS runningtotal
  FROM Sales.OrderValues
)
SELECT *
FROM RunningTotals
WHERE runningtotal < 1000.00;
```

This query generates the following output (shown here in abbreviated form):

```
custid      orderid     orderdate  val     runningtotal
----------- ----------- ---------- ------- -------------
1           10643       2015-08-25 814.50  814.50
2           10308       2014-09-18 88.80   88.80
2           10625       2015-08-08 479.75  568.55
2           10759       2015-11-28 320.00  888.55
3           10365       2014-11-27 403.20  403.20
...
```

As another example for a window frame extent, if you wanted the frame to include only the last three rows, you would use the form ROWS BETWEEN 2 PRECEDING AND CURRENT ROW.

As for the RANGE window frame extent, according to standard SQL, it allows you to define delimiters based on an offset from the current row's ordering value, as opposed to an offset in terms of a number of rows. However, T-SQL has a limited implementation of the RANGE option, supporting only UNBOUNDED (PRECEDING and FOLLOWING) and CURRENT ROW as delimiters. One subtle difference between ROWS and RANGE when using the same delimiters is that the former doesn't include peers (tied rows in terms of the ordering values) and the latter does.

IMPORTANT **IMPLICIT RANGE**

The ROWS option usually gets optimized much better than RANGE when using the same delimiters. If you define a window with a window order clause but without a window frame clause, the default is RANGE BETWEEN UNBOUNDED PRECEDING AND CURRENT ROW. Therefore, unless you are after the special behavior you get from RANGE that includes peers, make sure you explicitly use the ROWS option. Even if you do need RANGE, it might be a good idea to mention it explicitly to let everyone know that your choice was intentional and not an oversight. SQL Server 2016 eliminates this performance problem when using columnstore technology and the batch mode Window Aggregate operator. For details, see "What You Need to Know about the Batch Mode Window Aggregate Operator in SQL Server 2016: Part 3" at *http://sqlmag.com/sql-server/what-you-need-know-about-batch-mode-window-aggregate-operator-sql-server-2016-part-3*.

Window Ranking Functions

With window ranking functions, you can rank rows within a partition based on specified ordering. As with the other window functions, if you don't indicate a window partition clause, the entire underlying query result is considered one partition. The window order clause is mandatory. Window ranking functions do not support a window frame clause. T-SQL supports four window ranking functions: ROW_NUMBER, RANK, DENSE_RANK, and NTILE.

The following query demonstrates the use of these functions:

```
SELECT custid, orderid, val,
  ROW_NUMBER() OVER(ORDER BY val) AS rownum,
  RANK()       OVER(ORDER BY val) AS rnk,
  DENSE_RANK() OVER(ORDER BY val) AS densernk,
  NTILE(100)   OVER(ORDER BY val) AS ntile100
FROM Sales.OrderValues;
```

This query generates the following output (shown here in abbreviated form):

```
custid  orderid  val    rownum  rnk  densernk  ntile100
------- -------- ------ ------- ---- --------- ---------
12      10782    12.50  1       1    1         1
27      10807    18.40  2       2    2         1
66      10586    23.80  3       3    3         1
76      10767    28.00  4       4    4         1
54      10898    30.00  5       5    5         1
88      10900    33.75  6       6    6         1
48      10883    36.00  7       7    7         1
41      11051    36.00  8       7    7         1
71      10815    40.00  9       9    8         1
38      10674    45.00  10      10   9         2
53      11057    45.00  11      10   9         2
75      10271    48.00  12      12   10        2
...
```

IMPORTANT **ORDERING OF RESULT**

The sample query doesn't have a presentation ORDER BY clause, and therefore, there's no assurance that the rows will be presented in any particular order. The window order clause only determines ordering for the window function's computation. If you invoke a window function in your query but don't specify a presentation ORDER BY clause, there's no guarantee that the rows will be presented in the same order as the window function's ordering. If you need such a guarantee, you need to add a presentation ORDER BY clause.

The ROW_NUMBER function computes unique incrementing integers starting with 1 within the window partition based on the window ordering. Because the example query doesn't have a window partition clause, the function considers the entire query's result set as one partition; hence, the function assigns unique row numbers across the entire query's result set.

Note that if the ordering isn't unique, the ROW_NUMBER function is not deterministic. For example, notice in the result that two rows have the same ordering value of 36.00, but the

two rows were assigned with different row numbers. That's because the function must generate unique integers in the partition. Currently, there's no explicit tiebreaker, and therefore the choice of which row gets the higher row number is arbitrary (optimization dependent). If you need a deterministic computation (guaranteed repeatable results), you need to add a tiebreaker. For example, you can add the primary key to make the ordering unique, as in ORDER BY `val`, `orderid`.

RANK and DENSE_RANK differ from ROW_NUMBER in the sense that they assign the same ranking value to all rows that share the same ordering value. The RANK function returns the number of rows in the partition that have a lower ordering value than the current, plus 1. For example, consider the rows in the sample query's result that have an ordering value of 45.00. Nine rows have ordering values that are lower than 45.00; hence, these rows got the rank 10 (9 + 1).

The DENSE_RANK function returns the number of distinct ordering values that are lower than the current, plus 1. For example, the same rows that got the rank 10 got the dense rank 9. That's because these rows have an ordering value 45.00, and there are eight distinct ordering values that are lower than 45.00. Because RANK considers the count of rows and DENSE_RANK considers the count of distinct values, the former can have gaps between result ranking values, and the latter cannot have gaps. Because the RANK and DENSE_RANK functions compute the same ranking value to rows with the same ordering value, both functions are deterministic even when the ordering isn't unique. In fact, if you use unique ordering, both functions return the same result as the ROW_NUMBER function. So usually these functions are interesting to use when the ordering isn't unique.

> ***NOTE*** **FUNCTION DETERMINISM**
>
> **The official T-SQL documentation has a section describing function determinism at *https://msdn.microsoft.com/en-us/library/ms178091.aspx*. You will notice that this article considers all window ranking functions as nondeterministic. That's because it seems that SQL Server doesn't consider the partitioning and ordering values technically as the function's inputs. With all window ranking functions, two different rows can get two different rank values, and therefore are considered nondeterministic. This means you can't create indexes on computed columns and indexed views when using such functions. The aspect of determinism that I described is a bit different and not related to restrictions on indexing. In my use of the concept of determinism I was referring to whether two rows with the same partitioning and ordering values are guaranteed to get the same rank value or not.**

With the NTILE function, you can arrange the rows within the partition in a requested number of equally sized tiles, based on the specified ordering. You specify the desired number of tiles as input to the function. In the sample query, you requested 100 tiles. There are 830 rows in the result set, so the base tile size is 830 / 100 = 8 with a remainder of 30. Be-

cause there is a remainder of 30, the first 30 tiles are assigned with an additional row. Namely, tiles 1 through 30 will have nine rows each, and all remaining tiles (31 through 100) will have eight rows each. Observe in the result of this sample query that the first nine rows (according to val ordering) are assigned with tile number 1, then the next nine rows are assigned with tile number 2, and so on. Like ROW_NUMBER, the NTILE function isn't deterministic when the ordering isn't unique. If you need to guarantee determinism, you need to define unique ordering.

EXAM TIP

As explained in the discussion of window aggregate functions, window functions are allowed only in the SELECT and ORDER BY clauses of the query. If you need to refer to those in other clauses—for example, in the WHERE clause—you need to use a table expression such as a CTE. You invoke the window function in the inner query's SELECT clause, and assign the expression with a column alias. Then you refer to that column alias in the outer query's WHERE clause. That's a common need with ranking calculations. I demonstrated this earlier in Skill 2.2 when discussing table expressions.

Window Offset Functions

Window offset functions return an element from a single row that is in a given offset from the current row in the window partition, or from the first or last row in the window frame. T-SQL supports the following window offset functions: LAG, LEAD, FIRST_VALUE, and LAST_VALUE. The LAG and LEAD functions rely on an offset with respect to the current row, and the FIRST_VALUE and LAST_VALUE functions operate on the first or last row in the frame, respectively.

The LAG and LEAD functions support window partition and ordering clauses. They don't support a window frame clause. The LAG function returns an element from the row in the current partition that is a requested number of rows before the current row (based on the window ordering), with 1 assumed as the default offset. The LEAD function returns an element from the row that is in the requested offset after the current row.

As an example, the following query uses the LAG and LEAD functions to return along with each order the value of the previous customer's order, in addition to the value from the next customer's order:

```
SELECT custid, orderid, orderdate, val,
  LAG(val)  OVER(PARTITION BY custid
                 ORDER BY orderdate, orderid) AS prev_val,
  LEAD(val) OVER(PARTITION BY custid
                  ORDER BY orderdate, orderid) AS next_val
FROM Sales.OrderValues;
```

This query generates the following output (shown here in abbreviated form, with presentation ordering not guaranteed):

```
custid      orderid     orderdate  val     prev_val  next_val
----------- ----------- ---------- ------- --------- ---------
1           10643       2015-08-25 814.50  NULL      878.00
1           10692       2015-10-03 878.00  814.50    330.00
1           10702       2015-10-13 330.00  878.00    845.80
1           10835       2016-01-15 845.80  330.00    471.20
1           10952       2016-03-16 471.20  845.80    933.50
1           11011       2016-04-09 933.50  471.20    NULL
2           10308       2014-09-18 88.80   NULL      479.75
2           10625       2015-08-08 479.75  88.80     320.00
2           10759       2015-11-28 320.00  479.75    514.40
2           10926       2016-03-04 514.40  320.00    NULL
...
```

Because an explicit offset wasn't specified, both functions relied on the default offset of 1. If you want a different offset than 1, you specify it as the second argument, as in LAG(val, 3). Notice that if a row does not exist in the requested offset, the function returns a NULL by default. If you want to return a different value in such a case, specify it as the third argument, as in LAG(val, 3, 0).

The FIRST_VALUE and LAST_VALUE functions return a value expression from the first or last rows in the window frame, respectively. Naturally, the functions support window partition, order, and frame clauses. As an example, the following query returns along with each order the values of the customer's first and last orders:

```
SELECT custid, orderid, orderdate, val,
  FIRST_VALUE(val)  OVER(PARTITION BY custid
                         ORDER BY orderdate, orderid
                         ROWS BETWEEN UNBOUNDED PRECEDING
                                  AND CURRENT ROW) AS first_val,
  LAST_VALUE(val) OVER(PARTITION BY custid
                       ORDER BY orderdate, orderid
                       ROWS BETWEEN CURRENT ROW
                                AND UNBOUNDED FOLLOWING) AS last_val
FROM Sales.OrderValues
ORDER BY custid, orderdate, orderid;
```

This query generates the following output (shown here in abbreviated form):

```
custid  orderid  orderdate  val     first_val  last_val
------- -------- ---------- ------- ---------- ---------
1       10643    2015-08-25 814.50  814.50     933.50
1       10692    2015-10-03 878.00  814.50     933.50
1       10702    2015-10-13 330.00  814.50     933.50
1       10835    2016-01-15 845.80  814.50     933.50
1       10952    2016-03-16 471.20  814.50     933.50
1       11011    2016-04-09 933.50  814.50     933.50
2       10308    2014-09-18 88.80   88.80      514.40
2       10625    2015-08-08 479.75  88.80      514.40
2       10759    2015-11-28 320.00  88.80      514.40
2       10926    2016-03-04 514.40  88.80      514.40
...
```

IMPORTANT **DEFAULT FRAME**

As a reminder, when a window frame is applicable to a function but you do not specify an explicit window frame clause, the default is RANGE BETWEEN UNBOUNDED PRECEDING AND CURRENT ROW. For performance reasons, it is generally recommended to avoid the RANGE option; to do so, you need to be explicit with the ROWS clause. Also, if you're after the first row in the partition, using the FIRST_VALUE function with the default frame at least gives you the correct result. However, if you're after the last row in the partition, using the LAST_VALUE function with the default frame won't give you what you want because the last row in the default frame is the current row. So with the LAST_VALUE, you need to be explicit about the window frame in order to get what you are after. And if you need an element from the last row in the partition, the second delimiter in the frame should be UNBOUNDED FOLLOWING.Note Book on Window Functions

For more detailed information about window functions, their optimization, and practical uses, refer to the book Microsoft SQL Server 2012 High-Performance T-SQL Using Window Functions, by Itzik Ben-Gan (Microsoft Press, 2012).

MORE INFO **ON ADVANCED USES OF WINDOW FUNCTIONS**

You can also find examples for advanced uses of window functions in a video recording of the session "Run, Total, Run!" at *https://www.youtube.com/watch?v=KM83eVqHHPA&feature=youtu.be*.

MORE INFO **ON STATISTICAL WINDOW FUNCTIONS**

For details and examples about statistical window functions known as *window distribution functions*, see "Microsoft SQL Server 2012: How to Write T-SQL Window Functions, Part 2 Using offset and distribution functions" at *http://sqlmag.com/sql-server-2012/microsoft-sql-server-2012-how-write-t-sql-window-functions-part-2*.

Skill 2.4: Query temporal data and non-relational data

This skill covers querying temporal data using system-versioned temporal tables. It also covers using T-SQL to query and output JSON data as well as to query and output XML data.

The sections about XML and JSON were written by Data Platform MVP, Dejan Sarka.

This section covers how to:

- Query historic data by using temporal tables
- Query and output JSON data
- Query and output XML data

System-versioned temporal tables

Companies often need to be able to track changes to their data. That's for purposes like auditing, point in time analysis, comparing current with older states, slowly-changing dimensions, restoring older state of rows due to an error, and others. Normally, you can only access the current state of the data in your tables. If you need to be able to access older states, you need a solution that tracks changes to the data. One of the solutions that companies used in the past was to create a history table that keeps older states of modified rows and triggers that automatically write history data to that table whenever the current table is modified. In SQL Server 2016 and Azure SQL Database there's no need for this anymore thanks to the support for a feature called *system-versioned temporal tables*. I'll just use the term *temporal tables* in short.

SQL Server supports marking a table as a temporal table using an option called SYSTEM_VERSIONING and connecting it to a history table. When you modify data, you interact only with the current table, and SQL Server behind the scenes writes historical states of modified rows to the history table. Also when you read data, you query only the current table. You use a clause called FOR SYSTEM_TIME that allows you to request earlier states of the data at a previous point or period of time.

At the date of writing SQL Server currently supports only system-versioned temporal tables, meaning that the system transaction time determines the effective time of the change. The SQL Standard also supports what's called *application-time period tables* where the application defines the validity period of a row. With this feature, you can set a change to be effective in a future period. For example, suppose that there's a planned price change of a product during an upcoming holiday period. Then bi-temporal tables combine system versioning and application versioning.

The following sections cover creating, modifying and querying temporal tables.

Creating tables

You can mark a table as a temporal table when you create it, or alter an existing table to become a temporal table. Also, you can have SQL Server create the related history table for you, or provide an already existing history table.

There are certain elements that are required in the table definition in order to mark it as a temporal one:

- A primary key constraint.

- Two DATETIME2 columns with your chosen precision to store the start and end of the validity period of the row (stored in the UTC time zone). The period is expressed as a closed-open interval, meaning that the start is inclusive and the end is exclusive.
- The start column needs to be marked with the clause GENERATED ALWAYS AS ROW START.
- The end column needs to be marked with the clause GENERATED ALWAYS AS ROW END.
- The designation of the pair of columns that store the row's validity period with the clause PERIOD FOR SYSTEM_TIME (<startcol>, <endcol>).
- The table option SYSTEM_VERSIONING needs to be set to ON.
- A linked history table, which SQL Server can create for you.

As an example, run the following code to create a table called dbo.Products in the TSQLV4 database as a temporal table (not to be confused with the already existing Production.Products table):

```
USE TSQLV4;

CREATE TABLE dbo.Products
(
  productid   INT          NOT NULL
    CONSTRAINT PK_dboProducts PRIMARY KEY(productid),
  productname NVARCHAR(40) NOT NULL,
  supplierid  INT          NOT NULL,
  categoryid  INT          NOT NULL,
  unitprice   MONEY        NOT NULL,
-- below are additions related to temporal table
  validfrom   DATETIME2(3)
    GENERATED ALWAYS AS ROW START HIDDEN NOT NULL,
  validto     DATETIME2(3)
    GENERATED ALWAYS AS ROW END   HIDDEN NOT NULL,
  PERIOD FOR SYSTEM_TIME (validfrom, validto)
)
WITH ( SYSTEM_VERSIONING = ON ( HISTORY_TABLE = dbo.ProductsHistory ) );
```

Observe the use of the optional HIDDEN property for the period columns. With this property, the period columns are not returned when using SELECT * rather only when referring to them explicitly.

Regarding the history table, if you don't specify one at all, SQL Server creates it for you with the naming convention: MSSQL_TemporalHistoryFor_<object_id>. If you do specify a history table as in the above example, SQL Server first checks if it already exists. If it does, SQL Server by default applies a consistency check to verify that there are no overlapping periods. You can opt not to perform the consistency check by specifying DATA_CONSISTENCY_CHECK = OFF. If the specified history table doesn't exist, SQL Server will create it for you using your chosen name. SQL Server creates the history table with the following characteristics:

- No primary key.
- A clustered index on (<endcol>, <startcol>), with page compression.

- The period columns are not marked with GENERATED ALWAYS AS ROW START/END or HIDDEN.
- There's no designation of period columns with the clause PERIOD FOR SYSTEM_TIME.
- The table is not marked with the option SYSTEM_VERSIONING.

You can see that a table is a temporal table in SQL Server Management Studio (SSMS). In Object Explorer, expand the Tables folder under the TSQLV4 database, and then the dbo.Products table, as shown in Figure 2-5.

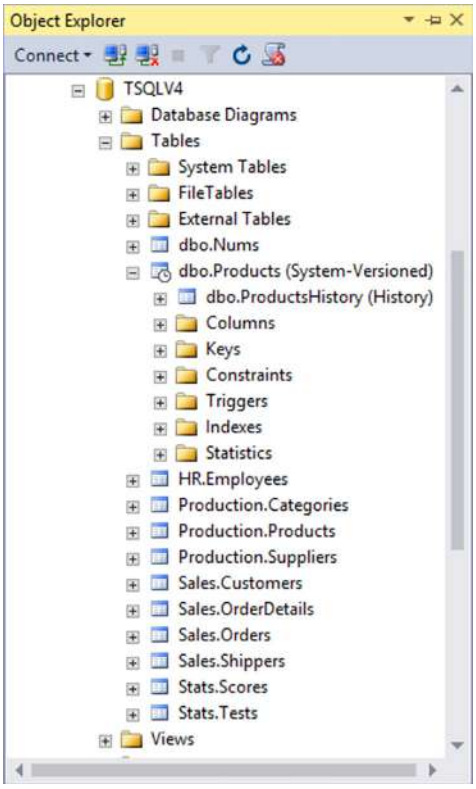


FIGURE 2-5 Object Explorer

Observe the table with the clock icon to the left of the current table name and the fact that it says System-Versioned in parentheses after the table name. Also, observe the connected history table below the current table, and the fact that it says History in parentheses after the history table name.

If you need to turn an existing table to become a temporal table, you do so by first altering the table and adding the period columns with the aforementioned designations, and then altering the table to mark it as system-versioned and connecting it to a history table. You will need to add default constraints to the period columns to set initial values in the existing rows. The end column in the current table has to store the maximum possible value in the data type with the select precision. Once the table is marked as a temporal table, you can drop the de-

fault constraints if you wish since SQL Server will automatically set the period column values moving forward. For example, if the dbo.Products wasn't already a temporal table, you would have turned it to become one by using the following code (don't actually run this code since our table is already temporal):

```
BEGIN TRAN;

ALTER TABLE dbo.Products ADD
  validfrom DATETIME2(3) GENERATED ALWAYS AS ROW START HIDDEN NOT NULL
    CONSTRAINT DFT_Products_validfrom DEFAULT('19000101'),
  validto DATETIME2(3) GENERATED ALWAYS AS ROW END HIDDEN NOT NULL
    CONSTRAINT DFT_Products_validto DEFAULT('99991231 23:59:59.999'),
  PERIOD FOR SYSTEM_TIME (validfrom, validto);

ALTER TABLE dbo.Products
  SET ( SYSTEM_VERSIONING = ON ( HISTORY_TABLE = dbo.ProductsHistory ) );

ALTER TABLE dbo.Products DROP CONSTRAINT DFT_Products_validfrom, DFT_Products_validto;

COMMIT TRAN;
```

SQL Server supports altering temporal tables. You alter the current table and SQL Server takes care of applying the relevant changes to the history table for you. For instance, suppose that you want to add a column called discontinued to the Products table. You alter the current table by running the following code:

```
ALTER TABLE dbo.Products
  ADD discontinued BIT NOT NULL
    CONSTRAINT DFT_Products_discontinued DEFAULT(0);
```

SQL Server adds the same column to the history table with the specified default value 0 in existing rows, if such were present, but does not add a default constraint in the history table.

Similarly, if you want to drop a column, you alter only the current table, and SQL Server takes care of dropping it from the history table. You will want to start by dropping the default constraint, and then the column. For example, run the following code to drop the column discontinued from the Products table:

```
ALTER TABLE dbo.Products
  DROP CONSTRAINT DFT_Products_discontinued;

ALTER TABLE dbo.Products
  DROP COLUMN discontinued;
```

SQL Server also drops the column discontinued from the history table for you.

Modifying data

When you modify data in a temporal table, you do so just like with a normal table. You submit your changes to the current table, and SQL Server takes care of writing history rows to the history table when relevant. A couple of things to remember:

1. SQL Server records the change times in the UTC time zone.
2. If you apply multiple changes in a transaction, the transaction start time is considered the effective change time for all changes in the transaction.

When you insert rows into the current table, SQL Server sets the start column to the transaction's start time and the end time to the maximum possible point in time in the data type. No rows need to be written to the history table. I ran the examples in this section on November 11th, 2016, so all of the period columns will have this date in the examples. Suppose that at 14:07:26.263 (UTC) you run the following code:

```
INSERT INTO dbo.Products(productid, productname, supplierid, categoryid, unitprice)
  SELECT productid, productname, supplierid, categoryid, unitprice
  FROM Production.Products
  WHERE productid <= 10;
```

Query the current table after inserting the data:

```
SELECT productid, supplierid, unitprice, validfrom, validto
FROM dbo.Products;
```

Notice that in order to return the period columns, the query refers to them explicitly since these columns are marked as hidden in our table. This query generates the following output (of course, in this case the validfrom column will reflect the time I ran the insert):

```
productid  supplierid  unitprice  validfrom                  validto
---------- ----------- ---------- -------------------------- -------------------------
1          1           18.00      2016-11-01 14:07:26.263  9999-12-31 23:59:59.999
2          1           19.00      2016-11-01 14:07:26.263  9999-12-31 23:59:59.999
3          1           10.00      2016-11-01 14:07:26.263  9999-12-31 23:59:59.999
4          2           22.00      2016-11-01 14:07:26.263  9999-12-31 23:59:59.999
5          2           21.35      2016-11-01 14:07:26.263  9999-12-31 23:59:59.999
6          3           25.00      2016-11-01 14:07:26.263  9999-12-31 23:59:59.999
7          3           30.00      2016-11-01 14:07:26.263  9999-12-31 23:59:59.999
8          3           40.00      2016-11-01 14:07:26.263  9999-12-31 23:59:59.999
9          4           97.00      2016-11-01 14:07:26.263  9999-12-31 23:59:59.999
10         4           31.00      2016-11-01 14:07:26.263  9999-12-31 23:59:59.999
```

As mentioned, when you insert data, only the current table is affected. Query the history table:

```
SELECT productid, unitprice, validfrom, validto
FROM dbo.ProductsHistory;
```

This query generates an empty set:

```
productid  supplierid  unitprice  validfrom                validto
---------- ----------- ---------- ------------------------ ------------------------
```

When you delete rows, SQL Server moves the affected rows to the history table, and sets the end column to the start time of the transaction that applied the change. Suppose that you run the following code to delete the row for product 10 at 14:08:41.758:

```
DELETE FROM dbo.Products
WHERE productid = 10;
```

An update is handled as a delete plus insert. In other words, the current table will have the new state of the modified rows, with the start column set to the change time, and the history table will have the old state of the modified rows, with the end column set to the change time. For example, suppose that you run the following code at 14:09:18.584:

```
UPDATE dbo.Products
  SET unitprice *= 1.05
WHERE supplierid = 3;
```

Products 6, 7 and 8 are affected.

Run the following code to query the current table after these changes:

```
SELECT productid, supplierid, unitprice, validfrom, validto
FROM dbo.Products;
```

This query generates the following output:

```
productid  supplierid  unitprice  validfrom                validto
---------- ----------- ---------- ------------------------ ------------------------
1          1           18.00      2016-11-01 14:07:26.263  9999-12-31 23:59:59.999
2          1           19.00      2016-11-01 14:07:26.263  9999-12-31 23:59:59.999
3          1           10.00      2016-11-01 14:07:26.263  9999-12-31 23:59:59.999
4          2           22.00      2016-11-01 14:07:26.263  9999-12-31 23:59:59.999
5          2           21.35      2016-11-01 14:07:26.263  9999-12-31 23:59:59.999
6          3           26.25      2016-11-01 14:09:18.584  9999-12-31 23:59:59.999
7          3           31.50      2016-11-01 14:09:18.584  9999-12-31 23:59:59.999
8          3           42.00      2016-11-01 14:09:18.584  9999-12-31 23:59:59.999
9          4           97.00      2016-11-01 14:07:26.263  9999-12-31 23:59:59.999
```

Observe that product 10 isn't present in the table, and products 6, 7 and 8, which are supplied by supplier 3, show the new prices, and a validity period starting at 2016-11-01 14:09:18.584.

Query the history table:

```
SELECT productid, supplierid, unitprice, validfrom, validto
FROM dbo.ProductsHistory;
```

This query generates the following output:

```
productid  supplierid  unitprice  validfrom                validto
---------- ----------- ---------- ------------------------ ------------------------
10         4           31.00      2016-11-01 14:07:26.263  2016-11-01 14:08:41.758
6          3           25.00      2016-11-01 14:07:26.263  2016-11-01 14:09:18.584
7          3           30.00      2016-11-01 14:07:26.263  2016-11-01 14:09:18.584
8          3           40.00      2016-11-01 14:07:26.263  2016-11-01 14:09:18.584
```

Observe that the old states of both deleted and updated rows are recorded in the history table, and notice their validity periods start with the insert time and end with the delete/update time.

As mentioned, when you modify data in a temporal table, the transaction start time is considered the effective time of the change. If you have multiple modifications within the same user transaction, they will all have the same effective modification time. For example, consider the following code, which modifies three different rows within a single transaction, with five second intervals between the modifications:

```
BEGIN TRAN;

PRINT CAST(SYSUTCDATETIME() AS DATETIME2(3));

UPDATE dbo.Products
  SET unitprice *= 0.95
WHERE productid = 1;

WAITFOR DELAY '00:00:05.000';

UPDATE dbo.Products
  SET unitprice *= 0.90
WHERE productid = 2;

WAITFOR DELAY '00:00:05.000';

UPDATE dbo.Products
  SET unitprice *= 0.85
WHERE productid = 3;

COMMIT TRAN;
```

The PRINT statement reported the following time as the transaction start time when running this code on my system:

```
2016-11-01 14:10:43.470
```

Run the following code to query the modified rows from the current table:

```
SELECT productid, supplierid, unitprice, validfrom, validto
FROM dbo.Products
WHERE productid IN (1, 2, 3);
```

This query generates the following output:

```
productid  supplierid  unitprice  validfrom                  validto
---------- ----------- ---------- -------------------------- -------------------------
1          1           17.10      2016-11-01 14:10:43.470  9999-12-31 23:59:59.999
2          1           17.10      2016-11-01 14:10:43.470  9999-12-31 23:59:59.999
3          1           8.50       2016-11-01 14:10:43.470  9999-12-31 23:59:59.999
```

Notice that all three rows show the same validfrom value, which is the transaction start time reported earlier. The older state of the modified rows was written to the history table with the validto value set to the transaction start time.

If you update the same row multiple times in the same transaction, a curious thing happens. The original and last states of the row will naturally have nonzero length intervals as the validity period; however, the in-between states will have *degenerate intervals* as the validity period where the validfrom value will be equal to the validto value. That's the result of using the transaction start time as the effective time of all changes in the transaction. Since the validity period is a closed-open interval, conceptually, it's as if those states lasted zero time. If you query the history table directly, you will see those rows that have a degenerate interval as the validity period. If you query the temporal table using the FOR SYSTEM_TIME clause to ask for a previous point or period of time, SQL Server will discard those rows.

As an example, run the following code to update the same row multiple times in the same transaction:

```
BEGIN TRAN;

PRINT CAST(SYSUTCDATETIME() AS DATETIME2(3));

UPDATE dbo.Products
  SET unitprice = 1.0
WHERE productid = 9;

WAITFOR DELAY '00:00:05.000';

UPDATE dbo.Products
  SET unitprice = 2.0
WHERE productid = 9;

WAITFOR DELAY '00:00:05.000';

UPDATE dbo.Products
  SET unitprice = 3.0
WHERE productid = 9;

COMMIT TRAN;
```

The PRINT statement reported the following transaction start time when I ran this code on my system:

```
2016-11-01 14:11:38.113
```

Query the current table:

```
SELECT productid, unitprice, validfrom, validto
FROM dbo.Products
WHERE productid = 9;
```

This query generates the following output, showing the current state:

```
productid  unitprice  validfrom                validto
---------- ---------- ------------------------ ------------------------
9          3.00       2016-11-01 14:11:38.113  9999-12-31 23:59:59.999
```

Query history table:

```
SELECT productid, unitprice, validfrom, validto
FROM dbo.ProductsHistory
WHERE productid = 9;
```

This query generates the following output, showing that two of the states have a degenerate interval as the validity period:

```
productid  unitprice  validfrom                validto
---------- ---------- ------------------------ ------------------------
9          97.00      2016-11-01 14:07:26.263  2016-11-01 14:11:38.113
9          1.00       2016-11-01 14:11:38.113  2016-11-01 14:11:38.113
9          2.00       2016-11-01 14:11:38.113  2016-11-01 14:11:38.113
```

Querying data

You can directly query the current and history tables, and get current and historical states of the rows. However, to simplify the code, SQL Server enables you to ask for the state of the data at a previous point or period of time by querying only the current table and specifying a clause called FOR SYSTEM_TIME. Using different subclauses you can specify either a point or a period during which the rows were valid. Behind the scenes, SQL Server will retrieve the data from both the current and history tables as needed, and return a single unified set of rows as the result set of the query. You can also query views with the FOR SYSTEM_TIME clause, provided that there's at least one underlying table that is a temporal table. SQL Server will propagate the clause to the underlying temporal tables.

If you want to run the code samples in this section and get the same results as the ones I'll provide, you need to populate the tables with the same data as in mine. Run the following code to achieve this:

```
USE TSQLV4;

-- drop tables if exist
IF OBJECT_ID(N'dbo.Products', N'U') IS NOT NULL
BEGIN
  IF OBJECTPROPERTY(OBJECT_ID(N'dbo.Products', N'U'), N'TableTemporalType') = 2
    ALTER TABLE dbo.Products SET ( SYSTEM_VERSIONING = OFF );
  DROP TABLE IF EXISTS dbo.ProductsHistory, dbo.Products;
END;
GO
```

```
-- create and populate Products table
CREATE TABLE dbo.Products
(
  productid   INT          NOT NULL
    CONSTRAINT PK_dboProducts PRIMARY KEY(productid),
  productname NVARCHAR(40) NOT NULL,
  supplierid  INT          NOT NULL,
  categoryid  INT          NOT NULL,
  unitprice   MONEY        NOT NULL,
  validfrom   DATETIME2(3) NOT NULL,
  validto     DATETIME2(3) NOT NULL
);

INSERT INTO dbo.Products
  (productid, productname, supplierid, categoryid, unitprice, validfrom, validto)
VALUES
  (1, 'Product HHYDP', 1, 1, 17.10, '20161101 14:10:43.470', '99991231 23:59:59.999'),
  (2, 'Product RECZE', 1, 1, 17.10, '20161101 14:10:43.470', '99991231 23:59:59.999'),
  (3, 'Product IMEHJ', 1, 2,  8.50, '20161101 14:10:43.470', '99991231 23:59:59.999'),
  (4, 'Product KSBRM', 2, 2, 22.00, '20161101 14:07:26.263', '99991231 23:59:59.999'),
  (5, 'Product EPEIM', 2, 2, 21.35, '20161101 14:07:26.263', '99991231 23:59:59.999'),
  (6, 'Product VAIIV', 3, 2, 26.25, '20161101 14:09:18.584', '99991231 23:59:59.999'),
  (7, 'Product HMLNI', 3, 7, 31.50, '20161101 14:09:18.584', '99991231 23:59:59.999'),
  (8, 'Product WVJFP', 3, 2, 42.00, '20161101 14:09:18.584', '99991231 23:59:59.999'),
  (9, 'Product AOZBW', 4, 6,  3.00, '20161101 14:11:38.113', '99991231 23:59:59.999');

-- create and populate ProductsHistory table
CREATE TABLE dbo.ProductsHistory
(
  productid   INT          NOT NULL,
  productname NVARCHAR(40) NOT NULL,
  supplierid  INT          NOT NULL,
  categoryid  INT          NOT NULL,
  unitprice   MONEY        NOT NULL,
  validfrom   DATETIME2(3) NOT NULL,
  validto     DATETIME2(3) NOT NULL,
  INDEX ix_ProductsHistory CLUSTERED(validto, validfrom)
    WITH (DATA_COMPRESSION = PAGE)
);

INSERT INTO dbo.ProductsHistory
  (productid, productname, supplierid, categoryid, unitprice, validfrom, validto)
VALUES
  ( 1, 'Product HHYDP', 1, 1, 18.00, '20161101 14:07:26.263', '20161101 14:10:43.470'),
  ( 2, 'Product RECZE', 1, 1, 19.00, '20161101 14:07:26.263', '20161101 14:10:43.470'),
  ( 3, 'Product IMEHJ', 1, 2, 10.00, '20161101 14:07:26.263', '20161101 14:10:43.470'),
  ( 6, 'Product VAIIV', 3, 2, 25.00, '20161101 14:07:26.263', '20161101 14:09:18.584'),
  ( 7, 'Product HMLNI', 3, 7, 30.00, '20161101 14:07:26.263', '20161101 14:09:18.584'),
  ( 8, 'Product WVJFP', 3, 2, 40.00, '20161101 14:07:26.263', '20161101 14:09:18.584'),
  ( 9, 'Product AOZBW', 4, 6, 97.00, '20161101 14:07:26.263', '20161101 14:11:38.113'),
  ( 9, 'Product AOZBW', 4, 6,  1.00, '20161101 14:11:38.113', '20161101 14:11:38.113'),
  ( 9, 'Product AOZBW', 4, 6,  2.00, '20161101 14:11:38.113', '20161101 14:11:38.113'),
  (10, 'Product YHXGE', 4, 8, 31.00, '20161101 14:07:26.263', '20161101 14:08:41.758');

-- enable system versioning
```

```
ALTER TABLE dbo.Products ADD PERIOD FOR SYSTEM_TIME (validfrom, validto);

ALTER TABLE dbo.Products ALTER COLUMN validfrom ADD HIDDEN;
ALTER TABLE dbo.Products ALTER COLUMN validto ADD HIDDEN;

ALTER TABLE dbo.Products
  SET ( SYSTEM_VERSIONING = ON ( HISTORY_TABLE = dbo.ProductsHistory ) );
```

Query the current table to see its contents:

```
SELECT productid, supplierid, unitprice, validfrom, validto
FROM dbo.Products;
```

This query generates the following output:

```
productid  supplierid  unitprice  validfrom                  validto
---------- ----------- ---------- -------------------------- -------------------------
1          1           17.10      2016-11-01 14:10:43.470  9999-12-31 23:59:59.999
2          1           17.10      2016-11-01 14:10:43.470  9999-12-31 23:59:59.999
3          1           8.50       2016-11-01 14:10:43.470  9999-12-31 23:59:59.999
4          2           22.00      2016-11-01 14:07:26.263  9999-12-31 23:59:59.999
5          2           21.35      2016-11-01 14:07:26.263  9999-12-31 23:59:59.999
6          3           26.25      2016-11-01 14:09:18.584  9999-12-31 23:59:59.999
7          3           31.50      2016-11-01 14:09:18.584  9999-12-31 23:59:59.999
8          3           42.00      2016-11-01 14:09:18.584  9999-12-31 23:59:59.999
9          4           3.00       2016-11-01 14:11:38.113  9999-12-31 23:59:59.999
```

Query the history table to see its contents:

```
SELECT productid, supplierid, unitprice, validfrom, validto
FROM dbo.ProductsHistory;
```

This query generates the following output:

```
productid  supplierid  unitprice  validfrom                  validto
---------- ----------- ---------- -------------------------- -------------------------
10         4           31.00      2016-11-01 14:07:26.263  2016-11-01 14:08:41.758
6          3           25.00      2016-11-01 14:07:26.263  2016-11-01 14:09:18.584
7          3           30.00      2016-11-01 14:07:26.263  2016-11-01 14:09:18.584
8          3           40.00      2016-11-01 14:07:26.263  2016-11-01 14:09:18.584
1          1           18.00      2016-11-01 14:07:26.263  2016-11-01 14:10:43.470
2          1           19.00      2016-11-01 14:07:26.263  2016-11-01 14:10:43.470
3          1           10.00      2016-11-01 14:07:26.263  2016-11-01 14:10:43.470
9          4           97.00      2016-11-01 14:07:26.263  2016-11-01 14:11:38.113
9          4           1.00       2016-11-01 14:11:38.113  2016-11-01 14:11:38.113
9          4           2.00       2016-11-01 14:11:38.113  2016-11-01 14:11:38.113
```

A commonly used subclause of the FOR SYSTEM_TIME clause is AS OF @dt. It returns the rows that were valid during the input point in time @dt. Since the validity period is a closed-open interval, you get the rows where @dt >= validfrom AND @dt < validto (@dt is on or after validfrom and before validto). In other words, the validity period starts on or before @dt and ends after @dt.

Recall that I issued the first insertion into the Products table at 14:07:26.263. Run the following code to query the state of the data at a point in time prior to the first insertion:

```
SELECT productid, supplierid, unitprice
FROM dbo.Products FOR SYSTEM_TIME AS OF '20161101 14:06:00.000';
```

You get an empty set back.

Run the following code to query the state of the data after the first insertion and before any other modification:

```
SELECT productid, supplierid, unitprice
FROM dbo.Products FOR SYSTEM_TIME AS OF '20161101 14:07:55.000';
```

This query generates the following output:

```
productid  supplierid  unitprice
---------- ----------- ----------
4          2           22.00
5          2           21.35
10         4           31.00
6          3           25.00
7          3           30.00
8          3           40.00
1          1           18.00
2          1           19.00
3          1           10.00
9          4           97.00
```

You can also query multiple instances of the table, each with a different point in time as input to the AS OF clause, and this way compare different states of the data at different points in time. For example, the following query identifies products that experienced an increase in the unit price between the points 14:08:55 and 14:10:55, and returns the percent of increase in the price:

```
SELECT T1.productid, T1.productname,
  CAST( (T2.unitprice / T1.unitprice - 1.0) * 100.0 AS NUMERIC(10, 2) ) AS pct
FROM dbo.Products FOR SYSTEM_TIME AS OF '20161101 14:08:55.000' AS T1
  INNER JOIN dbo.Products FOR SYSTEM_TIME AS OF '20161101 14:10:55.000' AS T2
    ON T1.productid = T2.productid
   AND T2.unitprice > T1.unitprice;
```

This query generates the following output:

```
productid  productname    pct
---------- -------------- -----
6          Product VAIIV  5.00
7          Product HMLNI  5.00
8          Product WVJFP  5.00
```

It would appear that three products experience a five percent increase in their price.

The second subclause of the FOR SYSTEM_TIME clause is FROM @start TO @end. It returns all rows that have a validity period that intersects with the input period, exclusive of the two input delimiters. You get the rows where validfrom < @end AND validto > @start. Namely

the validity interval starts before the input interval ends and ends after the input interval starts. As mentioned, the FOR SYSTEM_TIME clause discards degenerate intervals with all subclauses.

As an example, the following query returns the versions of product 9 that were valid during a period that ended after 2016-11-01 14:00:00.000 and started before 2016-11-01 14:11:38.113 (intersection excluding the edges):

```
SELECT productid, supplierid, unitprice, validfrom, validto
FROM dbo.Products
  FOR SYSTEM_TIME FROM '20161101 14:00:00.000' TO '20161101 14:11:38.113'
WHERE productid = 9;
```

This query generates the following output:

```
productid  supplierid  unitprice  validfrom                validto
---------- ----------- ---------- ------------------------ ------------------------
9          4           97.00      2016-11-01 14:07:26.263  2016-11-01 14:11:38.113
```

The output shows that there's only one version that qualifies. The product row experienced a change at 2016-11-01 14:11:38.113, but since both input delimiters are excluded, that version isn't returned. In order to include the input end time, use the subclause BETWEEN @start AND @end instead. You get the rows where validfrom <= @end AND validto > @start. Namely, the validity interval starts *on or* before the input interval ends and ends after the input interval starts.

Here's the same query as the last, only this time using BETWEEN instead of FROM:

```
SELECT productid, supplierid, unitprice, validfrom, validto
FROM dbo.Products
  FOR SYSTEM_TIME BETWEEN '20161101 14:00:00.000' AND '20161101 14:11:38.113'
WHERE productid = 9;
```

This time the query returns two versions of the row for product 9:

```
productid  supplierid  unitprice  validfrom                validto
---------- ----------- ---------- ------------------------ ------------------------
9          4           3.00       2016-11-01 14:11:38.113  9999-12-31 23:59:59.999
9          4           97.00      2016-11-01 14:07:26.263  2016-11-01 14:11:38.113
```

The CONTAINED IN(@start, @end) subclause returns rows with a validity period that is entirely contained within the input period, inclusive of both input delimiters. You get the rows where validfrom >= @start AND validto <= @end. Meaning, that the validity interval starts on or after the input interval starts and ends on or before the input interval ends.

As an example, the following code returns the rows with a validity that is contained in the period that starts with 2016-11-01 14:07:00.000 and ends with 2016-11-01 14:10:00.000:

```
SELECT productid, supplierid, unitprice, validfrom, validto
FROM dbo.Products
  FOR SYSTEM_TIME CONTAINED IN('20161101 14:07:00.000', '20161101 14:10:00.000');
```

This query generates the following output:

```
productid   supplierid  unitprice   validfrom                   validto
----------- ----------- ----------- --------------------------- --------------------------
10          4           31.00       2016-11-01 14:07:26.263     2016-11-01 14:08:41.758
6           3           25.00       2016-11-01 14:07:26.263     2016-11-01 14:09:18.584
7           3           30.00       2016-11-01 14:07:26.263     2016-11-01 14:09:18.584
8           3           40.00       2016-11-01 14:07:26.263     2016-11-01 14:09:18.584
```

What's interesting is that you would expect the SQL Server optimizer to realize that since all rows in the current table have a validity period that ends with the maximum value in the type, there's no need to internally access the current table at all. But that's not the case. Examine the execution plan that SQL Server creates for this query as shown in Figure 2-6.

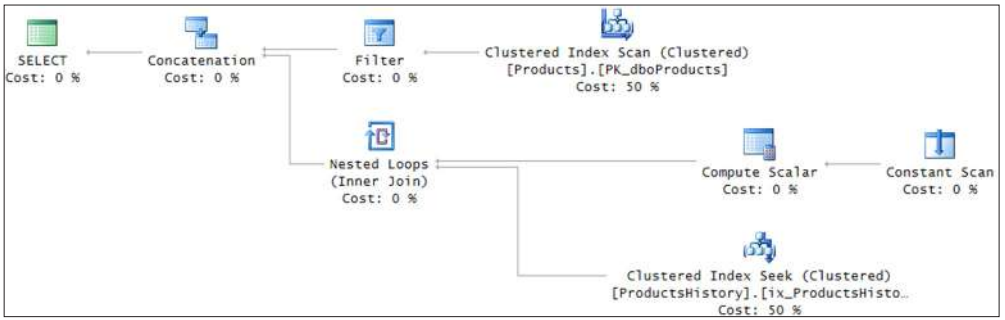


FIGURE 2-6 Query plan when CHECK constraint doesn't exist

Notice that both tables are accessed.

What you can do to help the optimizer figure out it doesn't need to access the current table is to add a CHECK constraint that verifies that the `validto` column value is indeed the maximum possible value in the type, like so:

```
ALTER TABLE dbo.Products
  ADD CONSTRAINT CHK_Products_validto
    CHECK (validto = '99991231 23:59:59.999');
```

Rerun the CONTAINED IN query after adding the constraint and examine the plan for this query as shown in Figure 2-7.

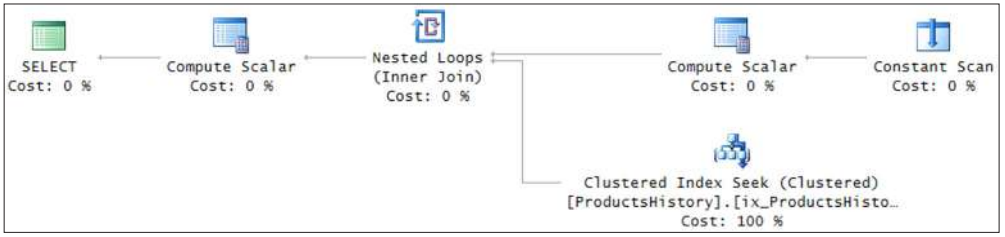


FIGURE 2-7 Query plan when CHECK constraint exists

This time based on a contradiction detection the optimizer realizes that it doesn't need to access the current table.

The ALL subclause simply returns all versions, from both the current and history table, excluding degenerate intervals. As an example, the following query returns all versions of all rows from the Products table:

```
SELECT productid, supplierid, unitprice, validfrom, validto
FROM dbo.Products FOR SYSTEM_TIME ALL
ORDER BY productid, validfrom, validto;
```

This query generates the following output:

```
productid  supplierid  unitprice  validfrom                validto
---------- ----------- ---------- ------------------------ ------------------------
1          1           18.00      2016-11-01 14:07:26.263  2016-11-01 14:10:43.470
1          1           17.10      2016-11-01 14:10:43.470  9999-12-31 23:59:59.999
2          1           19.00      2016-11-01 14:07:26.263  2016-11-01 14:10:43.470
2          1           17.10      2016-11-01 14:10:43.470  9999-12-31 23:59:59.999
3          1           10.00      2016-11-01 14:07:26.263  2016-11-01 14:10:43.470
3          1           8.50       2016-11-01 14:10:43.470  9999-12-31 23:59:59.999
4          2           22.00      2016-11-01 14:07:26.263  9999-12-31 23:59:59.999
5          2           21.35      2016-11-01 14:07:26.263  9999-12-31 23:59:59.999
6          3           25.00      2016-11-01 14:07:26.263  2016-11-01 14:09:18.584
6          3           26.25      2016-11-01 14:09:18.584  9999-12-31 23:59:59.999
7          3           30.00      2016-11-01 14:07:26.263  2016-11-01 14:09:18.584
7          3           31.50      2016-11-01 14:09:18.584  9999-12-31 23:59:59.999
8          3           40.00      2016-11-01 14:07:26.263  2016-11-01 14:09:18.584
8          3           42.00      2016-11-01 14:09:18.584  9999-12-31 23:59:59.999
9          4           97.00      2016-11-01 14:07:26.263  2016-11-01 14:11:38.113
9          4           3.00       2016-11-01 14:11:38.113  9999-12-31 23:59:59.999
10         4           31.00      2016-11-01 14:07:26.263  2016-11-01 14:08:41.758
```

Remember that the period columns store the time in the UTC time zone. If you want to present the values in a desired target time zone, you can use the AT TIME ZONE function to achieve this. There are a couple of things to think about, though. One is that you need one conversion from a non-offset type to an offset type using <period_column> AT TIME ZONE 'UTC', and another conversion to switch the UTC value to the target time zone. Another is that when the `validto` value is the maximum in the type, you don't want to switch its offset, but rather keep it as UTC. You can use a CASE expression to achieve this. As an example, the following query returns all versions of rows from the Products table and presents the period column values in the time zone Pacific Standard Time:

```
SELECT productid, unitprice,
  validfrom AT TIME ZONE 'UTC' AT TIME ZONE 'Pacific Standard Time' AS validfrom,
  CASE
    WHEN validto = '99991231 23:59:59.999'
      THEN validto AT TIME ZONE 'UTC'
    ELSE validto AT TIME ZONE 'UTC' AT TIME ZONE 'Pacific Standard Time'
  END AS validto
FROM dbo.Products FOR SYSTEM_TIME ALL
ORDER BY productid, validfrom, validto;
```

This query generates the following output:

```
productid  unitprice  validfrom                      validto
---------- ---------- ------------------------------ ------------------------------
1          18.00      2016-11-01 07:07:26.263 -07:00 2016-11-01 07:10:43.470 -07:00
1          17.10      2016-11-01 07:10:43.470 -07:00 9999-12-31 23:59:59.999 +00:00
2          19.00      2016-11-01 07:07:26.263 -07:00 2016-11-01 07:10:43.470 -07:00
2          17.10      2016-11-01 07:10:43.470 -07:00 9999-12-31 23:59:59.999 +00:00
3          10.00      2016-11-01 07:07:26.263 -07:00 2016-11-01 07:10:43.470 -07:00
3          8.50       2016-11-01 07:10:43.470 -07:00 9999-12-31 23:59:59.999 +00:00
4          22.00      2016-11-01 07:07:26.263 -07:00 9999-12-31 23:59:59.999 +00:00
5          21.35      2016-11-01 07:07:26.263 -07:00 9999-12-31 23:59:59.999 +00:00
6          25.00      2016-11-01 07:07:26.263 -07:00 2016-11-01 07:09:18.584 -07:00
6          26.25      2016-11-01 07:09:18.584 -07:00 9999-12-31 23:59:59.999 +00:00
7          30.00      2016-11-01 07:07:26.263 -07:00 2016-11-01 07:09:18.584 -07:00
7          31.50      2016-11-01 07:09:18.584 -07:00 9999-12-31 23:59:59.999 +00:00
8          40.00      2016-11-01 07:07:26.263 -07:00 2016-11-01 07:09:18.584 -07:00
8          42.00      2016-11-01 07:09:18.584 -07:00 9999-12-31 23:59:59.999 +00:00
9          97.00      2016-11-01 07:07:26.263 -07:00 2016-11-01 07:11:38.113 -07:00
9          3.00       2016-11-01 07:11:38.113 -07:00 9999-12-31 23:59:59.999 +00:00
10         31.00      2016-11-01 07:07:26.263 -07:00 2016-11-01 07:08:41.758 -07:00
```

When you're done, run the following code for cleanup:

```
IF OBJECT_ID(N'dbo.Products', N'U') IS NOT NULL
BEGIN
  IF OBJECTPROPERTY(OBJECT_ID(N'dbo.Products', N'U'), N'TableTemporalType') = 2
    ALTER TABLE dbo.Products SET ( SYSTEM_VERSIONING = OFF );
  DROP TABLE IF EXISTS dbo.ProductsHistory, dbo.Products;
END;
```

Query and output XML data

XML is a widely used standard for data exchange, that calls Web services methods, configuration files, and more. This section starts with a short introduction to XML. After that, you learn how to create XML as a result of a query by using different flavors of the FOR XML clause. The section finishes with information on shredding XML to relational tables by using the OPENXML `rowset` function.

The following is an example of an XML document, created with the FOR XML clause of the SELECT statement:

```
<CustomersOrders>
  <Customer custid="1" companyname="Customer NRZBB">
    <Order orderid="10692" orderdate="2007-10-03T00:00:00" />
    <Order orderid="10702" orderdate="2007-10-13T00:00:00" />
    <Order orderid="10952" orderdate="2008-03-16T00:00:00" />
  </Customer>
  <Customer custid="2" companyname="Customer MLTDN">
    <Order orderid="10308" orderdate="2006-09-18T00:00:00" />
    <Order orderid="10926" orderdate="2008-03-04T00:00:00" />
  </Customer>
</CustomersOrders>
```

As you can see, XML uses *tags* to name parts of an *XML document*. These parts are called *elements*. Every begin tag, such as <Customer>, must have a corresponding end tag, in this case </Customer>. If an element has no nested elements, the notation can be abbreviated to a single tag that denotes the beginning and end of an element, such as <Order … />. Elements can be nested, and tags cannot be interleaved; the end tag of a parent element must be after the end tag of the last nested element. If every begin tag has a corresponding end tag, and if tags are nested properly, the XML document is *well-formed*.

XML documents are *ordered*. This does not mean they are ordered by any specific element value; it means that the position of elements matters. For example, the element with `orderid` equal to 10702 in the preceding example is the second Order element under the first Customer element.

XML is *case-sensitive Unicode text*. You should never forget that XML is case sensitive. In addition, some characters in XML, such as <, which introduces a tag, are processed as *markup* and have special meanings. If you want to include these characters in the values of your XML document, they must be escaped using an ampersand (&), followed by a special code, followed by a semicolon (;), as shown in Table 2-1.

TABLE 2-1 Characters with special values in XML documents

Character	Replacement text
& (ampersand)	&amp;
" (quotation mark)	&quot;
< (less than)	&lt;
> (greater than)	&gt;
' (apostrophe)	&apos;

Alternatively, you can use the special XML CDATA section written as <![CDATA[...]]>. You can replace the three dots with any character string that does not include the string literal "]]>"; this will prevent special characters in the string from being parsed as XML markup.

XML can have a prolog at the beginning of the document, denoting the XML version and encoding of the document, such as <?xml version="1.0" encoding="ISO-8859-15"?>.

In addition to XML documents, you can also have *XML fragments*. The only difference between a document and a fragment is that a document has a single *root node*, like <CustomersOrders> in the preceding example. If you delete this node, you get an XML fragment.

As you can see from the example, elements can have *attributes*. Attributes have their own names, and their values are enclosed in quotes. This is *attribute-centric* presentation. However, you can write XML differently; every attribute can be a nested element of the original element. This is *element-centric* presentation.

Element names do not have to be unique, because they can be referred to by their position; however, to distinguish between elements from different business areas, different departments, or different companies, you can add *namespaces*. You declare namespaces used in

the root element of an XML document. You can also use an alias for every single namespace. Then you prefix element names with a namespace *alias*.

XML is very flexible. There are very few rules for creating a well-formed XML document. In an XML document, the actual data is mixed with *metadata*, such as element and attribute names. Because XML is text, it is very convenient for exchanging data between different systems and even between different platforms. However, when exchanging data, it becomes important to have metadata fixed. If you had to import a document with customers' orders, as in the preceding examples, every couple of minutes, you'd definitely want to automate the import process. Imagine how hard you'd have to work if the metadata changed with every new import.

Many different standards have evolved to describe the metadata of XML documents. Currently, the most widely used metadata description is with *XML Schema Description* (XSD) documents. XSD documents are XML documents that describe the metadata of other XML documents. The schema of an XSD document is predefined. With the XSD standard, you can specify element names, data types, and number of occurrences of an element, constraints, and more.

> ***MORE INFO*** **ON XML SCHEMAS**
>
> **For details about XML schemas, see the MSDN article "Understanding XML Schema" at *https://msdn.microsoft.com/en-us/library/aa468557.aspx*.**

When you check whether an XML document complies with a schema, you *validate* the document. A document with a predefined schema is said to be a *typed* XML document.

Producing and using XML in queries

With the T-SQL SELECT statement, you can create all of the XML outputs that are shown in this section. This section explains how you can convert a query result set to XML by using the FOR XML clause of the SELECT statement. Here you will learn about the most useful options and directives of this clause.

> ***MORE INFO*** **ON THE FOR XML CLAUSE**
>
> **For additional information about the FOR XML clause, please refer to the Books Online for SQL Server 2016 article "FOR XML (SQL Server)" at *https://msdn.microsoft.com/en-us/library/ms178107.aspx*.**

The first option for creating XML from a query result is the RAW option. The XML created is quite close to the relational (tabular) presentation of the data. In RAW mode, every row from the returned result set converts to a single element named row, and columns translate to the attributes of this element. Here is an example of a query that creates the RAW version of the FOR XML output:

```
SELECT Customer.custid, Customer.companyname,
```

```
  [Order].orderid, [Order].orderdate
FROM Sales.Customers AS Customer
  INNER JOIN Sales.Orders AS [Order]
    ON Customer.custid = [Order].custid
WHERE Customer.custid <= 2
  AND [Order].orderid %2 = 0
ORDER BY Customer.custid, [Order].orderid
FOR XML RAW;
```

This query generates the following output:

```
<row custid="1" companyname="Customer NRZBB" orderid="10692" orderdate="2015-10-03" />
<row custid="1" companyname="Customer NRZBB" orderid="10702" orderdate="2015-10-13" />
<row custid="1" companyname="Customer NRZBB" orderid="10952" orderdate="2016-03-16" />
<row custid="2" companyname="Customer MLTDN" orderid="10308" orderdate="2014-09-18" />
<row custid="2" companyname="Customer MLTDN" orderid="10926" orderdate="2016-03-04" />
```

You can notice that the result is actually an XML fragment and not an XML document, because the root node is missing. You can enhance the RAW mode by renaming the row element, adding a root element, including namespaces, and making the XML returned element-centric.

The FOR XML AUTO option gives you nice XML documents with nested elements, and it is not complicated to use. In AUTO and RAW modes, you can use the keyword ELEMENTS to produce element-centric XML. The WITH NAMESPACES clause, preceding the SELECT part of the query, defines namespaces and aliases in the returned XML. Here is an example of a query with the FOR XML AUTO option used, element-centric, with a namespace defined:

```
WITH XMLNAMESPACES('ER70761-CustomersOrders' AS co)
SELECT [co:Customer].custid AS [co:custid],
  [co:Customer].companyname AS [co:companyname],
  [co:Order].orderid AS [co:orderid],
  [co:Order].orderdate AS [co:orderdate]
FROM Sales.Customers AS [co:Customer]
  INNER JOIN Sales.Orders AS [co:Order]
    ON [co:Customer].custid = [co:Order].custid
WHERE [co:Customer].custid <= 2
  AND [co:Order].orderid %2 = 0
ORDER BY [co:Customer].custid, [co:Order].orderid
FOR XML AUTO, ELEMENTS, ROOT('CustomersOrders');
```

The T-SQL table and column aliases in the query are used to produce element names, prefixed with a namespace. A colon is used in XML to separate the namespace from the element name. The WHERE clause of the query limits the output to two customers, with only every second order for each customer retrieved. The output is quite a nice element-centric XML document. This query produces the following result (shown here in abbreviated form):

```
<CustomersOrders xmlns:co="ER70761-CustomersOrders">
  <co:Customer>
    <co:custid>1</co:custid>
    <co:companyname>Customer NRZBB</co:companyname>
    <co:Order>
      <co:orderid>10692</co:orderid>
```

```
      <co:orderdate>2015-10-03</co:orderdate>
    </co:Order>
…
    <co:custid>2</co:custid>
    <co:companyname>Customer MLTDN</co:companyname>
    <co:Order>
…
    </co:Order>
  </co:Customer>
</CustomersOrders>
```

Note that a proper ORDER BY clause is very important. With the SELECT statement, you are actually formatting the returned XML. Without the ORDER BY clause, the order of rows returned is unpredictable, and you can get a weird XML document with an element repeated multiple times with just part of nested elements every time.

It is not only the ORDER BY clause that is important; the order of columns in the SELECT clause also influences the XML returned. SQL Server uses column order to determine the nesting of elements. The order of the columns should follow one-to-many relationships. A customer can have many orders; therefore, you should have customer columns before order columns in your query.

You might be vexed by the fact that you have to take care of column order; in a relation, the order of columns and rows is not important. Nevertheless, you have to realize that the result of your query is not a relation; it is text in XML format, and parts of your query are used for formatting the text.

There are additional more detailed options for formatting the returned XML document by using the FOR XML PATH option. In addition, besides the XML document, you can also return the XSD schema of the document with the XMLSCHEMA directive.

Besides generating XML from a query, you can also do the opposite: convert XML to tables. Converting XML to relational tables is known as *shredding* XML. You shred XML with the OPENXML `rowset` function.

The OPENXML function provides a rowset over in-memory XML documents by using the *Document Object Model* (DOM) presentation. Before parsing the DOM, you need to prepare it. To prepare the DOM presentation of XML, you need to call the system stored procedure sys.sp_xml_preparedocument. After you shred the document, you must remove the DOM presentation by using the system procedure sys.sp_xml_removedocument.

The OPENXML function uses the following parameters:

- An XML DOM document handle, returned by sp_xml_preparedocument
- An XPath expression to find the nodes you want to map to rows of a rowset returned
- A description of the rowset returned
- Mapping between XML nodes and rowset columns

The document handle is an integer. This is the simplest parameter. The XPath expression is specified as *rowpattern*, which defines how XML nodes translate to rows. The path to a node is used as a pattern; nodes below the selected node define rows of the returned rowset.

You can map XML elements or attributes to rows and columns by using the WITH clause of the OPENXML function. In this clause, you can specify an existing table, which is used as a template for the rowset returned, or you can define a table with syntax similar to the CREATE TABLE statement syntax.

The OPENXML function accepts an optional third parameter, called flags, which allows you to specify the mapping used between the XML data and the relational rowset. The value 1 means attribute-centric mapping, and 2 means element-centric. Flag value 8 can be combined with values 1 and 2 with a bitwise OR (|) operator to get both attribute and element-centric mapping. The XML used for the following OPENXML examples uses attributes and elements; for example, `custid` is the attribute and `companyname` is the element. The intention of this slightly overcomplicated XML is to show you the difference between attribute-centric and element-centric mappings. The following code shreds an XML by using 11 for the flag parameter (8 | 1 | 2).

```
DECLARE @DocHandle AS INT;
DECLARE @XmlDocument AS NVARCHAR(1000);
SET @XmlDocument = N'
<CustomersOrders>
  <Customer custid="1">
    <companyname>Customer NRZBB</companyname>
    <Order orderid="10692">
      <orderdate>2015-10-03T00:00:00</orderdate>
    </Order>
    <Order orderid="10702">
      <orderdate>2015-10-13T00:00:00</orderdate>
    </Order>
    <Order orderid="10952">
      <orderdate>2016-03-16T00:00:00</orderdate>
    </Order>
  </Customer>
  <Customer custid="2">
    <companyname>Customer MLTDN</companyname>
    <Order orderid="10308">
      <orderdate>2014-09-18T00:00:00</orderdate>
    </Order>
    <Order orderid="10926">
      <orderdate>2016-03-04T00:00:00</orderdate>
    </Order>
  </Customer>
</CustomersOrders>';
-- Create an internal representation
EXEC sys.sp_xml_preparedocument @DocHandle OUTPUT, @XmlDocument;
-- Attribute- and element-centric mapping
-- Combining flag 8 with flags 1 and 2
SELECT *
FROM OPENXML (@DocHandle, '/CustomersOrders/Customer', 11)
     WITH (custid INT,
           companyname NVARCHAR(40));
-- Remove the DOM
EXEC sys.sp_xml_removedocument @DocHandle;
```

This query produces the following result:

```
custid      companyname
----------- ----------------------------------------
1           Customer NRZBB
2           Customer MLTDN
```

> ***MORE INFO*** **ON OPENXML**
>
> **For more details about the OPENXML function, please refer to the "OPENXML (Transact-SQL)" topic at *https://msdn.microsoft.com/en-us/library/ms186918.aspx*.**

Querying XML data with XQuery

XQuery is a standard language for browsing XML instances and returning XML output. It is much richer than *XPath* expressions—an older standard, which you can use for simple navigation only. With XQuery, you can navigate as with XPath; however, you can also loop over nodes, shape the returned XML instance, and much more.

XQuery, like XML, is case sensitive. For example, if you write Data() instead of data(), you will get an error stating that there is no Data() function. XQuery returns sequences. Sequences can include *atomic* values or *complex* values (XML nodes). Any node, such as an element, attribute, text, processing instruction, comment, or document, can be included in the sequence. Of course, you can format the sequences to get well-formed XML.

Every identifier in XQuery is a qualified name, or a *QName*. A QName consists of a local name and, optionally, a namespace prefix. You define namespaces in the *prolog*, which appears at the beginning of your XQuery expression. You separate the prolog from the query body with a semicolon. In addition, in T-SQL, you can declare namespaces used in XQuery expressions in advance in the WITH clause of the SELECT statement. If your XML uses a single namespace, you can also declare it as the default namespace for all elements in the XQuery prolog.

XQuery uses about 50 predefined data types. Do not worry too much about XQuery types; you'll never use most of them. This paragraph lists only the most important ones, without going into details about them. XQuery data types are divided into node types and atomic types. The node types include attribute, comment, element, namespace, text, processing-instruction, and document-node. The most important atomic types you might use in queries are xs:boolean, xs:string, xs:QName, xs:date, xs:time, xs:datetime, xs:float, xs:double, xs:decimal and xs:integer. Just as there are many data types, there are dozens of functions in XQuery as well.

A basic way to navigate in the XML document using XQuery is with XPath expressions. With XQuery, you can specify a path absolutely or relatively from the current node. XQuery takes care of the current position in the document; it means you can refer to a path relatively, starting from current node, where you navigated through a previous path expression. Every

path consists of a sequence of steps, listed from left to right. A complete path might take the following form: `Node-name/child::element-name[@attribute-name=value]`.

The real power of XQuery lies in the so-called *FLWOR* expressions. FLWOR is the acronym for `for`, `let`, `where`, `order by`, and `return`. A FLWOR expression is actually a `for each` loop. You can use it to iterate through a sequence returned by an XPath expression. Although you typically iterate through a sequence of nodes, you can use FLWOR expressions to iterate through any sequence. You can limit the nodes to be processed with a predicate, sort the nodes, and format the returned XML. The parts of a FLWOR statement are:

- **For** With a `for` clause, you bind iterator variables to input sequences. Input sequences are either sequences of nodes or sequences of atomic values. You create atomic value sequences using literals or functions.
- **Let** With the optional `let` clause, you assign a value to a variable for a specific iteration. The expression used for an assignment can return a sequence of nodes or a sequence of atomic values.
- **Where** With the optional `where` clause, you filter the iteration.
- **Order by** Using the `order by` clause, you can control the order in which the elements of the input sequence are processed. You control the order based on atomic values.
- **Return** The `return` clause is evaluated once per iteration, and the results are returned to the client in the iteration order. With this clause, you format the resulting XML.

Here is an example of a SELECT statement with a complex XQuery expression that uses all of the five FLWOR expressions:

```
DECLARE @x AS XML = N'
<CustomersOrders>
  <Customer custid="1">
    <!-- Comment 111 -->
    <companyname>Customer NRZBB</companyname>
    <Order orderid="10692">
      <orderdate>2015-10-03T00:00:00</orderdate>
    </Order>
    <Order orderid="10702">
      <orderdate>2015-10-13T00:00:00</orderdate>
    </Order>
    <Order orderid="10952">
      <orderdate>2016-03-16T00:00:00</orderdate>
    </Order>
  </Customer>
  <Customer custid="2">
    <!-- Comment 222 -->
    <companyname>Customer MLTDN</companyname>
    <Order orderid="10308">
      <orderdate>2014-09-18T00:00:00</orderdate>
    </Order>
    <Order orderid="10952">
      <orderdate>2016-03-04T00:00:00</orderdate>
    </Order>
```

```
  </Customer>
</CustomersOrders>';
SELECT @x.query('for $i in CustomersOrders/Customer/Order
                 let $j := $i/orderdate
                 where $i/@orderid < 10900
                 order by ($j)[1]
                 return
                 <Order-orderid-element>
                  <orderid>{data($i/@orderid)}</orderid>
                  {$j}
                 </Order-orderid-element>')
       AS [Filtered, sorted and reformatted orders with let clause];
```

This query produces the following XML fragment:

```
<Order-orderid-element>
  <orderid>10308</orderid>
  <orderdate>2014-09-18T00:00:00</orderdate>
</Order-orderid-element>
<Order-orderid-element>
  <orderid>10692</orderid>
  <orderdate>2015-10-03T00:00:00</orderdate>
</Order-orderid-element>
<Order-orderid-element>
  <orderid>10702</orderid>
  <orderdate>2015-10-13T00:00:00</orderdate>
</Order-orderid-element>
```

MORE INFO **ON XQUERY**

For more details on XQuery, please refer to the MSDN topic "XQuery Language Reference (SQL Server)" at *https://msdn.microsoft.com/en-us/library/ms189075.aspx*.

The XML data type

In SQL Server, you can store XML as simple text. However, plain text representation means having no knowledge of the structure built into an XML document. You can decompose the text, store it in multiple relational tables, and use relational technologies to manipulate the data. Relational structures are quite static and not so easy to change. Think of dynamic or volatile XML structures. Storing XML data in a native XML data type solves these problems, enabling functionality attached to the type that can accommodate support for a wide variety of XML technologies. SQL Server supports a native XML data type.

In the XQuery introduction in this chapter, you saw the XML data type. The XQuery expression was a parameter for the `query()` method of this type. The XML data type supports five methods that accept an XQuery expression as a parameter. Those methods enable querying (the `query()` method), retrieving atomic values (the `value()` method), existence checks (the `exist()` method), modifying sections within the XML data (the `modify()` method) as opposed to overwriting the whole thing, and shredding XML data into multiple rows in a result set (the `nodes()` method).

The `value()` method of the XML data type returns a scalar value, so it can be specified anywhere where scalar values are allowed, for example in the SELECT list of a query. Note that the `value()` method accepts an XQuery expression as the first input parameter. The second parameter is the SQL Server data type returned. The `value()` method must return a scalar value; therefore, you have to specify the position of the element in the sequence you are browsing, even if you know that there is only one.

You can use the `exist()` method to test if a specific node exists in an XML instance. Typical usage of this clause is in the WHERE clause of T-SQL queries. The `exist()` method returns a bit, a flag that represents true or false. It can return the following:

- 1, representing true, if the XQuery expression in a query returns a nonempty result. That means that the node searched for exists in the XML instance.
- 0, representing false, if the XQuery expression returns an empty result.
- NULL, if the XML instance is NULL.

The `query()` method, as the name implies, is used to query XML data. You already know this method from an earlier example. It returns an instance of an untyped XML value.

The XML data type is a large object type. The amount of data stored in a column of this type can be very large. It would not be very practical to replace the complete value when all you need is just to change a small portion of it, for example, a scalar value of some subelement. The SQL Server XML data type provides you with the `modify()` method, similar in concept to the WRITE method that can be used in a T-SQL UPDATE statement for VARCHAR(MAX) and the other MAX types. You invoke the `modify()` method in an UPDATE T-SQL statement.

The W3C standard doesn't support data modification with XQuery. However, SQL Server provides its own language extensions to support data modification with XQuery. SQL Server XQuery supports three data modification language (DML) keywords for data modification: insert, delete, and replace value of.

The `nodes()` method is useful when you want to shred an XML value into relational data. Its purpose is therefore the same as the purpose of the OPENXML rowset function introduced earlier. However, using the `nodes()` method is usually much faster than preparing the DOM with a call to sp_xml_preparedocument, executing a SELECT ... FROM OPENXML statement, and calling sp_xml_removedocument. The `nodes()` method prepares DOM internally, during the execution of the SELECT statement. The OPENXML approach could be faster if you prepared the DOM once and then shredded it multiple times in the same batch.

The result of the `nodes()` method is a result set that contains logical copies of the original XML instances. In those logical copies, the context node of every row instance is set to one of the nodes identified by the XQuery expression, meaning that you get a row for every single node from the starting point defined by the XQuery expression. The `nodes()` method returns copies of the XML values, so you have to use additional methods to extract the scalar values out of them. The `nodes()` method has to be invoked for every row in the table. With the T-SQL APPLY operator, you can invoke a right table expression for every row of a left table expression in the FROM part.

You will learn how to use an XML data type inside your database through an example. This example shows how you can make a relational database schema dynamic. The example extends the Products table from the TSQLV4 database.

Suppose that you need to store some specific attributes only for beverages and other attributes only for condiments. For example, you need to store the percentage of recommended daily allowance (RDA) of vitamins only for beverages, and a short description only for condiments to indicate the condiment's general character (such as sweet, spicy, or salty). You could add an XML data type column to the Production.Products table; for this example, call it additionalattributes. Because the other product categories have no additional attributes, this column has to be nullable. The following code alters the Production.Products table to add this column:

```
ALTER TABLE Production.Products
 ADD additionalattributes XML NULL;
```

Before inserting data in the new column, you might want to constrain the values of this column. You should use a typed XML, which is an XML validated against a schema. With an XML schema, you constrain the possible nodes, the data type of those nodes, and more. In SQL Server, you can validate XML data against an XML schema collection. This is exactly what you need for a dynamic schema; if you could validate XML data against a single schema only, you could not use an XML data type for a dynamic schema solution, because XML instances would be limited to a single schema. Validation against a collection of schemas enables support of different schemas for beverages and condiments. If you wanted to validate XML values only against a single schema, you would define only a single schema in the collection.

You create the schema collection by using the CREATE XML SCHEMA COLLECTION T-SQL statement. You have to supply the XML schema, an XSD document, as input. Creating the schema is a task that should not be taken lightly. If you make an error in the schema, some invalid data might be accepted and some valid data might be rejected.

The easiest way to create XML schemas is to create relational tables first, and then use the XMLSCHEMA option of the FOR XML clause. Store the resulting XML value (the schema) in a variable, and provide the variable as input to the CREATE XML SCHEMA COLLECTION statement. The following code creates two auxiliary empty tables for beverages and condiments, and then uses a SELECT statement with the FOR XML clause to create an XML schema from those tables. Then it stores the schemas in a variable, and creates a schema collection from that variable. Finally, after the schema collection is created, the code drops the auxiliary tables.

```
-- Auxiliary tables
CREATE TABLE dbo.Beverages(percentvitaminsRDA INT);
CREATE TABLE dbo.Condiments(shortdescription NVARCHAR(50));
GO
-- Store the schemas in a variable and create the collection
DECLARE @mySchema AS NVARCHAR(MAX) = N'';
SET @mySchema +=
  (SELECT *
   FROM Beverages
```

```
   FOR XML AUTO, ELEMENTS, XMLSCHEMA('Beverages'));
SET @mySchema +=
  (SELECT *
   FROM Condiments
   FOR XML AUTO, ELEMENTS, XMLSCHEMA('Condiments'));
SELECT CAST(@mySchema AS XML);
CREATE XML SCHEMA COLLECTION dbo.ProductsAdditionalAttributes AS @mySchema;
GO
-- Drop auxiliary tables
DROP TABLE dbo.Beverages, dbo.Condiments;
GO
```

The next step is to alter the XML column from a well-formed state to a schema-validated one:

```
ALTER TABLE Production.Products
  ALTER COLUMN additionalattributes
   XML(dbo.ProductsAdditionalAttributes);
```

Before using the new data type, you have to take care of one more issue. How do you prevent binding the wrong schema to a product of a specific category? For example, how do you prevent binding a condiments schema to a beverage? You could solve this issue with a trigger; however, having a declarative constraint—a check constraint—is preferable. This is why the code added namespaces to the schemas. You need to check whether the namespace is the same as the product category name. You cannot use XML data type methods inside constraints. You have to create two additional functions: one retrieves the XML namespace of the additionalattributes XML column, and the other retrieves the category name of a product. In the check constraint, you can check whether the return values of both functions are equal. Here is the code that creates both functions and adds a check constraint to the Production.Products table:

```
-- Function to retrieve the namespace
CREATE FUNCTION dbo.GetNamespace(@chkcol AS XML)
 RETURNS NVARCHAR(15)
AS
BEGIN
 RETURN @chkcol.value('namespace-uri((/*)[1])','NVARCHAR(15)');
END;
GO
-- Function to retrieve the category name
CREATE FUNCTION dbo.GetCategoryName(@catid AS INT)
 RETURNS NVARCHAR(15)
AS
BEGIN
 RETURN
  (SELECT categoryname
   FROM Production.Categories
   WHERE categoryid = @catid);
END;
GO
-- Add the constraint
ALTER TABLE Production.Products ADD CONSTRAINT ck_Namespace
```

```
 CHECK (dbo.GetNamespace(additionalattributes) =
        dbo.GetCategoryName(categoryid));
GO
```

The infrastructure is prepared. Run the following code to try and insert some valid XML data into your new column:

```
-- Beverage
UPDATE Production.Products
   SET additionalattributes = N'
<Beverages xmlns="Beverages">
  <percentvitaminsRDA>27</percentvitaminsRDA>
</Beverages>'
WHERE productid = 1;
-- Condiment
UPDATE Production.Products
   SET additionalattributes = N'
<Condiments xmlns="Condiments">
  <shortdescription>very sweet</shortdescription>
</Condiments>'
WHERE productid = 3;
```

To test whether the schema validation and check constraint work, you should try to insert some invalid data as well:

```
-- String instead of int
UPDATE Production.Products
   SET additionalattributes = N'
<Beverages xmlns="Beverages">
  <percentvitaminsRDA>twenty seven</percentvitaminsRDA>
</Beverages>'
WHERE productid = 1;
-- Wrong namespace
UPDATE Production.Products
   SET additionalattributes = N'
<Condiments xmlns="Condiments">
  <shortdescription>very sweet</shortdescription>
</Condiments>'
WHERE productid = 2;
-- Wrong element
UPDATE Production.Products
   SET additionalattributes = N'
<Condiments xmlns="Condiments">
  <unknownelement>very sweet</unknownelement>
</Condiments>'
WHERE productid = 3;
```

You should get errors for all three UPDATE statements. You can check the data with the SELECT statement.

When you are done, run the following code for cleanup:

```
ALTER TABLE Production.Products
 DROP CONSTRAINT ck_Namespace;
ALTER TABLE Production.Products
 DROP COLUMN additionalattributes;
```

```
DROP XML SCHEMA COLLECTION dbo.ProductsAdditionalAttributes;
DROP FUNCTION dbo.GetNamespace;
DROP FUNCTION dbo.GetCategoryName;
```

> ***MORE INFO*** **ON THE XML DATA TYPE**
>
> **For more information on the XML data type, please refer to the MSDN topic "XML Data Type and Columns (SQL Server)" at *https://msdn.microsoft.com/en-us/library/hh403385.aspx*.**

The XML data type is actually a large object type. There can be up to 2 gigabytes (GB) of data in every single column value. Scanning through the XML data sequentially is not a very efficient way of retrieving a simple scalar value. With relational data, you can create an index on a filtered column, allowing an index seek operation instead of a table scan. Similarly, you can index XML columns with specialized *XML indexes*. The first index you create on an XML column is the *primary XML index*. This index contains a shredded persisted representation of the XML values. For each XML value in the column, the index creates several rows of data. The number of rows in the index is approximately the number of nodes in the XML value. Such an index alone can speed up searches for a specific element by using the `exist()` method. After creating the primary XML index, you can create up to three other types of *secondary XML indexes*:

- **PATH** This secondary XML index is especially useful if your queries specify path expressions. It speeds up the `exist()` method better than the primary XML index. Such an index also speeds up queries that use the `value()` method for a fully specified path.
- **VALUE** This secondary XML index is useful if queries are value-based and the path is not fully specified or it includes a wildcard.
- **PROPERTY** This secondary XML index is very useful for queries that retrieve one or more values from individual XML instances using the `value()` method.

The primary XML index has to be created first. It can be created only on tables with a clustered primary key.

> ***MORE INFO*** **ON XML INDEXES**
>
> **For details on how to create, use and maintain XML indexes, please refer to the MSDN topic "XML Indexes (SQL Server)" at *https://msdn.microsoft.com/en-us/library/ms191497.aspx*.**

Query and output JSON data

Although XML is a standard for many years, many developers do not like it because it is somehow too verbose. Especially when you use element-centric XML, you have each element name for every value listed twice in your XML document. In addition, XML is not very clear for reading. Don't understand this incorrectly: XML is here to stay. It is the standard for many things, for example for calling Web services, for storing configurations, for exchanging data,

and more. Nevertheless, a new simplified standard called JSON evolved in the last decade. JSON is simpler, easier to read than XML, focused on data exchange.

> ***NOTE*** **JSON SPECIFICATION**
>
> **You can read the full JSON specification in "The JSON Data Interchange Format" document at *http://www.ecma-international.org/publications/files/ECMA-ST/ECMA-404.pdf*.**

Let's start with a simple example again. The following document is showing the same data that was used for the first XML example earlier in this chapter, just this time in JSON format, produced with the FOR JSON clause of the T-SQL SELECT statement:

```
[
   {
      "custid":1,
      "companyname":"Customer NRZBB",
      "Order":[
         {
            "orderid":10692,
            "orderdate":"2015-10-03"
         },
         {
            "orderid":10702,
            "orderdate":"2015-10-13"
         },
         {
            "orderid":10952,
            "orderdate":"2016-03-16"
         }
      ]
   },
   {
      "custid":2,
      "companyname":"Customer MLTDN",
      "Order":[
         {
            "orderid":10308,
            "orderdate":"2014-09-18"
         },
         {
            "orderid":10926,
            "orderdate":"2016-03-04"
         }
      ]
   }
]
```

At first glance, you might think that JSON is even more verbose than XML. However, the number of characters is approximately the same as in the first XML example, which is attribute-centric, and is approximately 40% lower than it would be in an element-centric XML. In addition, the JSON data is formatted in a very extensive way, using as many lines as possible. SQL Server Management Studio (SSMS) does not format JSON in such a nice way as it does

XML. It returns JSON as a character string, which you can find at *https://jsonformatter.curious-concept.com/*.

You can also see from the previous example that JSON is readable and simple. A JSON object (or document) consists of *collections* of *name – value pairs* known as object *members*. The name and the value are separated by a *colon*. Object member separator is a *comma*. *Curly brackets* are wrapping collections. Name is always a string, while JSON supports only four *primitive data types* for values: string, number, Boolean, and null. In addition, a value can be of one of two *complex data types*: an array or a nested JSON object. Arrays are enclosed in brackets. Names don't need to be unique.

Like XML documents, JSON objects also use special characters, and you need to escape them using a backslash (\), followed by a special code, as shown in Table 2-2.

TABLE 2-2 Characters with special values in JSON objects

Character	Replacement text
" (quotation mark)	\"
\ (backslash)	\\
/ (slash)	\/
Backspace	\b
form feed	\f
new line	\l
carriage return	\r
horizontal tab	\t

In addition, the FOR JSON clause returns control characters (characters with ASCII code 00 to 31) in the JSON output in \u<code> format, where <code> is in hexadecimal format, for example CHAR(0) as \u0000 and CHAR(31) as u\001f.

Producing JSON output from queries

The first, simple option to generate JSON objects from T-SQL query results, is the FOR JSON AUTO clause. With this option, SQL Server formats the JSON output automatically, based on the order of columns in the SELECT list and on the order of tables in the FROM list. You can't change this format. The following query was used to produce the JSON output shown at the beginning of this section:

```
SELECT Customer.custid, Customer.companyname,
  [Order].orderid, [Order].orderdate
FROM Sales.Customers AS Customer
  INNER JOIN Sales.Orders AS [Order]
    ON Customer.custid = [Order].custid
WHERE Customer.custid <= 2
  AND [Order].orderid %2 = 0
ORDER BY Customer.custid, [Order].orderid
FOR JSON AUTO;
```

If you noticed, JSON does not support even date and time data types. Of course, the natural question is how SQL Server data types are converted to JSON data types. Table 2-3 shows the conversion rules.

TABLE 2-3 SQL Server to JSON data type conversion rules

Category	SQL Server type	JSON type
String	char, varchar, nchar, nvarchar	string
Numeric	tinyint, smallint, int, bigint, real, float, decimal, numeric	number
Boolean	Bit	Boolean – true or false
date & time	date, time, datetime, datetime2, datetimeoffset	string
Binary	BASE64-encoded string	BASE64-encoded string
CLR	user-defined CLR types, geometry, geography	not supported
Other	uniqueidentifier, money	string

You have much more influence on the format of the JSON returned with the FOR JSON PATH clause. In the PATH mode, you can wrap and nest objects using multiple levels of hierarchy. The following query is very simple, showing the basic usage of the PATH mode:

```
SELECT TOP (2) custid, companyname, contactname
FROM Sales.Customers
ORDER BY custid
FOR JSON PATH;
```

The formatted result is:

```
[
   {
      "custid":1,
      "companyname":"Customer NRZBB",
      "contactname":"Allen, Michael"
   },
   {
      "custid":2,
      "companyname":"Customer MLTDN",
      "contactname":"Hassall, Mark"
   }
]
```

Your first formatting option is to use dot-separated alias names for the column names. The dot-separated aliases produce nested objects. The following query shows the usage of dot separated alias; for the sake of brevity, it limits the result set to a single customer:

```
SELECT custid AS [CustomerId],
  companyname AS [Company],
  contactname AS [Contact.Name]
FROM Sales.Customers
WHERE custid = 1
FOR JSON PATH;
```

The formatted result is:

```
[
   {
      "CustomerId":1,
      "Company":"Customer NRZBB",
      "Contact":{
         "Name":"Allen, Michael"
      }
   }
]
```

Of course, you can combine data from multiple tables. Just take care to use the same first part of the member name (or the column alias) for the data you want to nest together, as shown in the following example:

```
SELECT c.custid AS [Customer.Id],
  c.companyname AS [Customer.Name],
  o.orderid AS [Order.Id],
  o.orderdate AS [Order.Date]
FROM Sales.Customers AS c
  INNER JOIN Sales.Orders AS o
    ON c.custid = o.custid
WHERE c.custid = 1
  AND o.orderid = 10692
ORDER BY c.custid, o.orderid
FOR JSON PATH;
```

Here is the result:

```
[
   {
      "Customer":{
         "Id":1,
         "Name":"Customer NRZBB"
      },
      "Order":{ '
         "Id":10692,
         "Date":"2015-10-03"
      }
   }
]
```

Of course, the next question is whether you could nest orders inside customers. You can use more than one dot in column aliases, like you would nest namespaces in a .NET application. The following query shows how to produce multiple nesting levels:

```
SELECT c.custid AS [Customer.Id],
  c.companyname AS [Customer.Name],
  o.orderid AS [Customer.Order.Id],
  o.orderdate AS [Customer.Order.Date]
FROM Sales.Customers AS c
  INNER JOIN Sales.Orders AS o
    ON c.custid = o.custid
WHERE c.custid = 1
```

```
  AND o.orderid = 10692
ORDER BY c.custid, o.orderid
FOR JSON PATH;
```

And here is the formatted result:

```
[
   {
      "Customer":{
         "Id":1,
         "Name":"Customer NRZBB",
         "Order":{
            "Id":10692,
            "Date":"2015-10-03"
         }
      }
   }
]
```

In the FOR JSON clause, in both AUTO and PATH modes, you can specify three additional clauses:

- **ROOT** Adds a single, top level member.
- **INCLUDE_NULL_VALUES** Include nulls in the output. Nulls are by default excluded from the JSON output.
- **WITHOUT_ARRAY_WRAPPER** Removes square brackets around the output.

The following query removes the array wrapper from the result:

```
SELECT c.custid AS [Customer.Id],
  c.companyname AS [Customer.Name],
  o.orderid AS [Customer.Order.Id],
  o.orderdate AS [Customer.Order.Date]
FROM Sales.Customers AS c
  INNER JOIN Sales.Orders AS o
    ON c.custid = o.custid
WHERE c.custid = 1
  AND o.orderid = 10692
ORDER BY c.custid, o.orderid
FOR JSON PATH,
    WITHOUT_ARRAY_WRAPPER;
```

The query returns the following result:

```
{
   "Customer":{
      "Id":1,
      "Name":"Customer NRZBB",
      "Order":{
         "Id":10692,
         "Date":"2015-10-03"
      }
   }
}
```

You cannot remove the array wrapper brackets and have a root element at the same time. The following query adds a root element instead of the brackets:

```
SELECT c.custid AS [Customer.Id],
  c.companyname AS [Customer.Name],
  o.orderid AS [Customer.Order.Id],
  o.orderdate AS [Customer.Order.Date]
FROM Sales.Customers AS c
  INNER JOIN Sales.Orders AS o
    ON c.custid = o.custid
WHERE c.custid = 1
  AND o.orderid = 10692
ORDER BY c.custid, o.orderid
FOR JSON PATH,
    ROOT('Customer 1');
```

Here's the result of this query:

```
{
   "Customer 1":[
      {
         "Customer":{
            "Id":1,
            "Name":"Customer NRZBB",
            "Order":{
               "Id":10692,
               "Date":"2015-10-03"
            }
         }
      }
   ]
}
```

Notice that the elements inside the root element are still treated as an array, and hence are enclosed in brackets.

Finally, the following query removes the array wrapper and includes nulls in the output:

```
SELECT c.custid AS [Customer.Id],
  c.companyname AS [Customer.Name],
  o.orderid AS [Customer.Order.Id],
  o.orderdate AS [Customer.Order.Date],
  NULL AS [Customer.Order.Delivery]
FROM Sales.Customers AS c
  INNER JOIN Sales.Orders AS o
    ON c.custid = o.custid
WHERE c.custid = 1
  AND o.orderid = 10692
ORDER BY c.custid, o.orderid
FOR JSON PATH,
    WITHOUT_ARRAY_WRAPPER,
    INCLUDE_NULL_VALUES;
```

Here's the result of this query:

```
{
```

```
   "Customer":{
      "Id":1,
      "Name":"Customer NRZBB",
      "Order":{
         "Id":10692,
         "Date":"2015-10-03",
         "Delivery":null
      }
   }
}
```

This should be enough to give you an idea how to format JSON output using the SELECT statement.

> ***MORE INFO*** **ON FOR JSON CLAUSE**
>
> **For more information about the FOR JSON clause, please read the MSDN article "Format Query Results as JSON with FOR JSON (SQL Server)" at *https://msdn.microsoft.com/en-us/library/dn921882.aspx*.**

Convert JSON data to tabular format

Of course, you don't just produce JSON from T-SQL queries, you can also read JSON data and present it in tabular format. Just like you shred XML with the OPENXML function, you shred JSON with the OPENJSON function. The function accepts two parameters: the JSON expression and the path where to start processing the JSON fragment. The second argument is optional; if you don't provide it, the whole JSON object is processed. The returned table can have an implicit schema, or you define an explicit schema in the WITH clause.

When you don't define an explicit schema, the OPENJSON function returns a table with three columns:

- **key** The name of the JSON property.
- **value** The actual value of the JSON property.
- **type** JSON data type of the value as a tiny integer. Table 2-4 shows the possible values of this column and their meaning.

TABLE 2-4 Characters with special values in JSON objects

Type value	JSON data type
0	null
1	string
2	number
3	true/false
4	array
5	object

Since the OPENJSON function can return only one table, it converts only the first level of JSON object properties as rows. Remember, a JSON object can have nested objects. The following code shows the simplest usage of the OPENJSON function:

```
DECLARE @json AS NVARCHAR(MAX) = N'
{
   "Customer":{
      "Id":1,
      "Name":"Customer NRZBB",
      "Order":{
         "Id":10692,
         "Date":"2015-10-03",
         "Delivery":null
      }
   }
}';
SELECT *
FROM OPENJSON(@json);
```

The result of this code is a single row for the Customer element (abbreviated):

```
key         value                                       type
----------- ------------------------------------------- ----
Customer    {         "Id":1,         "Name":"Custom... 5
```

As you can see, only the first level property is converted to a single row. To return properties of nested objects, you need to use the path parameter, like so:

```
DECLARE @json AS NVARCHAR(MAX) = N'
{
   "Customer":{
      "Id":1,
      "Name":"Customer NRZBB",
      "Order":{
         "Id":10692,
         "Date":"2015-10-03",
         "Delivery":null
      }
   }
}';
SELECT *
FROM OPENJSON(@json,'$.Customer');
```

This query returns all properties of the customer element, generating the following output:

```
key         value                                       type
----------- ------------------------------------------- ----
Id          1                                           2
Name        Customer NRZBB                              1
Order       {            "Id":10692,            "Dat... 5
```

In the path property, you can specify either *lax* or *strict* mode of checking for the JSON elements. Lax is the default and it means relaxed mode. If a property does not exist, you get

an empty table. In strict mode, if you refer to a non-existing property, you get an error. The following code shows this on a simple example:

```
DECLARE @json AS NVARCHAR(MAX) = N'
{
  "Customer":{
      "Name":"Customer NRZBB"
      }
}';
SELECT *
FROM OPENJSON(@json,'lax $.Buyer');
SELECT *
FROM OPENJSON(@json,'strict $.Buyer');
```

The first query returns an empty table, whereas the second query returns an empty table and an error.

Finally, I'll show how you can define an explicit schema in the WITH clause. The following query extracts customer id as integer, customer name as string, and customer orders as a nested JSON object of type NVARCHAR(MAX) from the JSON input:

```
DECLARE @json AS NVARCHAR(MAX) = N'
{
   "Customer":{
      "Id":1,
      "Name":"Customer NRZBB",
      "Order":{
         "Id":10692,
         "Date":"2015-10-03",
         "Delivery":null
      }
   }
}';
SELECT *
FROM OPENJSON(@json)
WITH
 (
 CustomerId   INT           '$.Customer.Id',
 CustomerName NVARCHAR(20)  '$.Customer.Name',
 Orders       NVARCHAR(MAX) '$.Customer.Order' AS JSON
);
```

Here is the result:

```
CustomerId CustomerName   Orders
---------- -------------- ------------------------------------------------------------
1          Customer NRZBB {            "Id":10692,            "Date":"2015-10-03",
```

> ***MORE INFO*** **ON THE OPENJSON FUNCTION**
>
> For more details on the OPENJSON function, please refer to the "OPENJSON (Transact-SQL)" MSDN topic at *https://msdn.microsoft.com/en-us/library/dn921885.aspx*.

In addition to shredding JSON data to rows with the OPENJSON rowset function, you can also extract a scalar value from JSON text with the JSON_VALUE function or an object or an array with the JSON_QUERY function. The following code shows how to use these two functions:

```
DECLARE @json AS NVARCHAR(MAX) = N'
{
   "Customer":{
      "Id":1,
      "Name":"Customer NRZBB",
      "Order":{
         "Id":10692,
         "Date":"2015-10-03",
         "Delivery":null
      }
   }
}';
SELECT JSON_VALUE(@json, '$.Customer.Id') AS CustomerId,
 JSON_VALUE(@json, '$.Customer.Name') AS CustomerName,
 JSON_QUERY(@json, '$.Customer.Order') AS Orders;
```

The query returns the same result as the previous query that used the OPENJSON function:

```
CustomerId CustomerName   Orders
---------- -------------- ------------------------------------------------------------
1          Customer NRZBB {            "Id":10692,            "Date":"2015-10-03",...
```

You can also update JSON text with the JSON_MODIFY function. You can update a scalar value of a property, add a new property and its value, add an element to an array, delete a property, and more. The function returns the modified JSON text. Here is an example of usage of this function:

```
DECLARE @json AS NVARCHAR(MAX) = N'
{
   "Customer":{
      "Id":1,
      "Name":"Customer NRZBB",
      "Order":{
         "Id":10692,
         "Date":"2015-10-03",
         "Delivery":null
      }
   }
}';
-- Update name
SET @json = JSON_MODIFY(@json, '$.Customer.Name', 'Modified first name');

-- Delete Id
SET @json = JSON_MODIFY(@json, '$.Customer.Id', NULL)

-- Insert last name
SET @json = JSON_MODIFY(@json, '$.Customer.LastName', 'Added last name')

PRINT @json;
```

The modified JSON document is shown below.

```
{
   "Customer":{

      "Name":"Modified first name",
      "Order":{
         "Id":10692,
         "Date":"2015-10-03",
         "Delivery":null
      }
   ,"LastName":"Added last name"}
}
```

For storing XML, SQL Server provides native XML data type. There is no native JSON data type. You can store JSON in a NVARCHAR(MAX) column. You can test the validity of the JSON document with the ISJSON function. The following example shows how to use this function:

```
SELECT ISJSON ('str') AS s1,  ISJSON ('') AS s2,
  ISJSON ('{}') AS s3,  ISJSON ('{"a"}') AS s4,
  ISJSON ('{"a":1}') AS s5;
```

And here is the result:

```
s1          s2          s3          s4          s5
----------- ----------- ----------- ----------- -----------
0           0           1           0           1
```

As you can see, JSON support in SQL Server is not yet a first-class citizen like XML, but you do have the means to both produce JSON data as a character string output from queries, as well as shred JSON data to tabular form.

Chapter summary

- Self-contained subqueries are independent of the outer query. They are convenient to troubleshoot since you can always highlight the inner query and execute it independently.
- Correlated subqueries have references to columns from the tables in the outer query and are generally more complex to work with.
- There are cases where SQL Server handles subqueries more efficiently then joins, and cases where the opposite is true. It's important to understand those cases, and also make sure that when performance is critical, you test both options and compare their performance.
- Using the APPLY operator you can apply a table expression to each row from some table. The CROSS APPLY operator doesn't return the left row if the right side is an empty set, whereas the OUTER APPLY operator does.
- Table expressions are named queries that help you simplify and reuse code. T-SQL supports four kinds of table expressions: derived tables, CTEs, views and inline table-valued functions.

- Use derived tables and CTEs when you need to use the table expression only in one statement. Use views and inline table-valued functions when you need to reuse the table expression. If there are no parameters involved, use views, otherwise use inline table-valued functions.
- Use table expressions when you don't want to persist the inner query's result in a work table, and temporary tables or table variables when you do.
- T-SQL supports both traditional grouped queries that define one grouping set, and grouped queries that define multiple grouping sets using the clauses GROUPING SETS, CUBE and ROLLUP. Use the GROUPING_ID function to compute a grouping set identifier.
- Use the PIVOT operator to pivot data from a state of rows to columns, and the UNPIVOT operator to unpivot data from a state of columns to rows. With PIVOT, make sure to use a table expression that project the elements involved to avoid grouping implicitly by undesired columns. With UNPIVOT, remember that the operator removes rows with NULLs.
- Window functions allow you to perform data analysis calculations against the underlying query result without hiding the detail. T-SQL supports aggregate, ranking, offset and statistical window functions.
- System-versioned temporal tables allow you to keep track of the history of changes to your data for long periods of time. You enable system versioning on a table, and connect it to a corresponding history table. Temporal tables use a pair of DATETIME2 columns to represent the start and end of the validity period of the row.
- You modify the current table as usual, and SQL Server keeps track of historical states of rows in the history table automatically.
- To read data, you query the current table with the FOR SYSTEM_TIME clause. Using the AT subclause you request to return the state of the data at a specified point in time. Using the FROM, BETWEEN and CONTAINED IN subclauses, you request to return the states of the data that were valid during a specified period of time. Using the ALL subclause, you request to see all states of the rows, both current and historical.
- You create XML documents as output of a SELECT statement with the FOR XML clause. You can read XML data and shred it to tabular format with the OPENXML function. You can store XML data in SQL Server in a column of the XML data type. You can use methods of this column to extract scalar values, XML fragments, modify XML data, and more. You use XQuery expressions as parameters to the XML data type methods to navigate to the appropriate element or attribute.
- There is no native JSON data type in SQL Server. However, you can create JSON documents from queries with the FOR JSON clause, shred JSON documents to tabular format with the OPENJSON function, extract scalar values and JSON fragments with the JSON_VALUE and JSON_QUERY functions, modify JSON data with the JSON_MODIFY function, and test validity with the ISJSON function.

Thought experiment

In this thought experiment, demonstrate your skills and knowledge of the topics covered in this chapter. You can find the answer to this thought experiment in the next section.

You're hired as a consultant to serve as a subject matter expert in a large database migration project. In one of the preliminary stages of the project you have a meeting with the company's DBAs, who pose a few questions. Demonstrate your expertise by answering the following questions:

1. Generally, when solving tasks with T-SQL, is it more efficient to use joins or subqueries?
2. What is the difference between a self-contained subquery and a correlated one?
3. In what way is the APPLY operator different than joins and subqueries? Can you provide an example when it should be used?
4. From a performance perspective, is it better to use table expressions or temporary tables?
5. What are the limitations of the PIVOT operator in T-SQL?
6. What are the limitations of the UNPIVOT operator in T-SQL?
7. What is the best way in T-SQL to compute running totals, and what are the performance considerations that you need to be aware of when using such computations?
8. Why can't you place a row number calculation in the WHERE clause if you want to filter a range of row numbers?
9. How would you quickly create, with minimal effort, an element-centric XML document from a T-SQL SELECT result set?
10. What is the difference between the JSON_VALUE and JSON_QUERY functions?

Thought experiment answer

This section contains the solution to the thought experiment.

1. You're not going to be perceived as a proper consultant unless you start your answer with "it depends." The answer to this question is no different. It's very rare in T-SQL that one tool is always better than another. But it is important to know what are the cases when each tool does better, and also how to tell which is better when you're not sure. Joins tend to do better when you need to apply multiple calculations, like aggregates, based on the same set of rows. With subqueries you access the data separately for each aggregate and with joins you access the data once for all aggregates. However, when looking for nonmatches, joins currently aren't optimized with the Anti Semi Join optimization, whereas subqueries are. With this optimization SQL Server short circuits the work as soon as a match is found, resulting typically in less effort.

When you're not certain which solution is better, test your solutions. The run time is ultimately what the user cares about. If there's a significant performance difference, you want to try and identify what SQL Server does differently by comparing the query execution plans.

2. When the inner query is completely independent of the outer query, it's a self-contained subquery. It can be highlighted and executed independently. A correlated subquery has references to columns from tables in the outer query. It cannot be run independently, making it harder to troubleshoot.
3. A join treats its two inputs as a set, therefore if a join input is a query (table expression), the query cannot refer to elements from the other input; in other words, a join doesn't support correlations. A subqueries can be correlated, but normally subqueries are limited to returning only one column. The APPLY operator combines the advantages of both joins and subqueries. Like in a join, if an input is a table expression, it can return multiple columns and multiple rows. Like a subquery, the right side can have correlations to elements from the left side. An example for a case where APPLY is handy is when you need to return the three most recent orders for each employee. You use the CROSS APPLY operator where the left input is the HR.Employees table and the right input is a correlated derived table where you use a TOP query against the Sales.Orders table, correlating the order's employee ID with the employee's employee ID.
4. Table expressions don't persist the result of the inner query physically anywhere. When you're querying a table expression, SQL Server inlines the inner query logic, and the physical processing interacts directly with the underlying table's data. Temporary tables and table variables do persist the result set that you store in them. So usually you want to use temporary tables when you have a result set that is expensive to create, and you need to interact with it multiple times. If you need to interact with the result only once, and the use of the temporary object is more about simplifying your solution or circumventing language limitations such as ones related to reuse of column aliases, you typically want to use table expressions in such cases.
5. The PIVOT operator is limited to only one aggregate. If you use the COUNT aggregate, you're limited to COUNT(*). In a static query, you have to hard code the spreading values. If you don't want to hard code those, you have to construct and execute the query string with dynamic SQL.
6. The UNPIVOT operator removes rows with NULLs in the value column. It doesn't make this step optional. It requires all columns that you're unpivoting to have exactly the same data type. It supports only one measure (values column). Like with PIVOT, in a static query you have to hard code the columns that you're unpivoting, or otherwise build and execute the query string with dynamic SQL.
7. The best way is with a window aggregate function with the frame ROWS UNBOUNDED PRECEDING. This is much more efficient than doing this with joins or subqueries. You can support the calculation with a POC index (partitioning, ordering, covering) to avoid the need for explicit sorting. You want to be careful not to use the RANGE option,

which is much more expensive. You want to remember that if you specify a window order clause but don't specify the window frame unit (ROWS or RANGE), you get RANGE by default. So, you need to make sure to explicitly use ROWS. If you use columnstore technology, the running total calculation can be optimized with a highly efficient batch mode Window Aggregate operator.

8. That's because as a window function, the ROW_NUMBER function is supposed to be applied to the underlying query result. In logical query processing terms, the underlying query result is established only when you get to the SELECT phase (step 5), after FROM, WHERE, GROUP BY and HAVING. For this reason, you can only use window functions in the SELECT and ORDER BY clauses of a query. If you need to refer to them in clauses that are processed before the SELECT clause, like the WHERE clause, write a query where you invoke the window function in the SELECT and assign it with an alias, and then have the outer query refer to that alias in the WHERE clause.
9. You should use the FOR XML AUTO, ELEMENTS, ROOT('Root Name') clause. You could use also FOR XML PATH, but this would not be with minimal effort. You need the ELEMENTS sub-clause to generate an element-centric XML. Finally, you need the ROOT sub-clause because without root you get an XML fragment and not an XML document.
10. You extract a scalar value from JSON text with the JSON_VALUE function and an object or an array with the JSON_QUERY function.

CHAPTER 3

Program databases by using Transact-SQL

This chapter covers programmability features in T-SQL. It starts with programmability objects like views, user-defined functions, and stored procedures. It then covers handling errors with the TRY-CATCH construct, and working with transactions. The chapter completes with coverage of handling of data types and treatment of NULLs.

Skills in this chapter:

- Create database programmability objects by using Transact-SQL
- Implement error handling and transactions
- Implement data types and NULLs

Skill 3.1: Create database programmability objects by using Transact-SQL

This skill focuses on working with programmability objects using T-SQL. It covers simplifying and reusing query logic using views; encapsulating single expressions, queries, and multiple statements in user-defined functions; and lastly, working with stored procedures.

NOTE **CREATE OR ALTER**

SQL Server 2016 starting with Service Pack 1 and Azure SQL Database support the CREATE OR ALTER command for views, user-defined functions, stored procedures, and triggers. This command creates the object if it doesn't exist and alters it if it already exists. Prior to the introduction of this command, it was up to you to handle logic, such as checking if the object already existed before dropping it with the DROP command, or using the DROP IF EXISTS command (introduced in SQL Server 2016 RTM), creating it with the CREATE command, or altering it with the ALTER command. Many examples in this skill use the CREATE OR ALTER command. If you're running this book's code samples on SQL Server 2016, make sure that you have at minimum Service Pack 1 installed because that's when this command was introduced. If you're using Azure SQL Database, there's nothing special that you need to do because the CREATE OR ALTER command is supported there.

This section covers how to:

- Create stored procedures, table-valued and scalar-valued user-defined functions, and views
- Implement input and output parameters in stored procedures
- Identify whether to use scalar-valued or table-valued functions
- Distinguish between deterministic and non-deterministic functions
- Create indexed views

Views

A view is a reusable named query, or *table expression*, whose definition is stored as an object in the database. It is accessible to users who were granted with permissions to query it. You can also modify data in underlying tables through it.

Views enable you to simplify the database's data model for users by presenting them with customized views of the data. For example, views can join underlying tables to simplify reporting queries. Instead of repeating complex queries in multiple places in the code, they can achieve reusability by querying the view. If you need to alter the structure of tables, you can use views to provide the application with backward compatible representation of the data. Views can also be used as a security layer, by granting users with access to the view but not to the underlying tables. This way, users can see only the customized representation of the data that you want them to see.

Working with views

There's probably no better way to start the discussion about working with views than by jumping straight to an example. The following code creates a view called Sales.OrderTotals in the TSQLV4 database:

```
USE TSQLV4;
GO
CREATE OR ALTER VIEW Sales.OrderTotals
  WITH SCHEMABINDING
AS

SELECT
  O.orderid, O.custid, O.empid, O.shipperid,  O.orderdate,
  O.requireddate, O.shippeddate,
  SUM(OD.qty) AS qty,
  CAST(SUM(OD.qty * OD.unitprice * (1 - OD.discount))
       AS NUMERIC(12, 2)) AS val
FROM Sales.Orders AS O
  INNER JOIN Sales.OrderDetails AS OD
    ON O.orderid = OD.orderid
GROUP BY
  O.orderid, O.custid, O.empid, O.shipperid, O.orderdate,
  O.requireddate, O.shippeddate;
GO
```

This view computes total order quantities and net values by joining the Sales.Orders and Sales.OrderDetails tables, grouping the data by the order elements, and aggregating the quantities and net values.

Notice the GO batch separator after the USE TSQLV4 statement. The CREATE OR ALTER VIEW statement must be the first statement in the batch. The same applies to the CREATE VIEW and ALTER VIEW statements.

Also, notice the use of the SCHEMABINDING option in the view's header. This option prevents structural changes to dependent tables and columns while the view exists. This option is not set by default, but there are situations where it's mandatory, such as if you want to create an index on the view. Some, including myself, see this option as a best practice that increases system stability since it helps you avoid having objects in the database that depend on nonexistent or altered objects. However, you do need to be aware that the use of SCHEMABINDING does increase the complexity of handling structural changes, such as ones done as part of application upgrades. Such changes require dropping and recreating schema-bound objects before and after the change, respectively.

Recall the discussion about table expressions in Chapter 2, Skill 2.2. I explained that a view is one of the kinds of table expressions that T-SQL supports in addition to derived tables, CTEs, and inline table-valued functions. I also explained that the inner query has to follow three requirements:

- All columns must have names. This means that if the column is a result of a computation, you must assign it with an alias.
- All column names must be unique. This means that if you join tables and you want to return columns with the same name from the different tables, you have to assign the columns with different aliases.
- The inner query is not allowed to have an ORDER BY clause, unless this clause supports a TOP or OFFSET-FETCH filter. Either way, unless the outer query against the view has its own ORDER BY clause, presentation ordering for the rows in the result is not guaranteed.

If you need to assign aliases to target columns, you can use an inline aliasing form where you assign the alias as part of the expression, such as in the example above for the result columns qty and val. As an alternative, you could use an external aliasing form where you specify the target column names right after the view name in parentheses, like so:

```
CREATE OR ALTER VIEW Sales.OrderTotals
  (orderid, custid, empid, shipperid,  orderdate, requireddate, shippeddate,
   qty, val)...
```

Run the following code to query the view that you just created:

```
SELECT orderid, orderdate, custid, empid, val
FROM Sales.OrderTotals;
```

You get the following output, shown here in abbreviated form:

```
orderid  orderdate  custid  empid  val
-------- ---------- ------- ------ --------
10248    2014-07-04 85      5      440.00
10249    2014-07-05 79      6      1863.40
10250    2014-07-08 34      4      1552.60
10251    2014-07-08 84      3      654.06
10252    2014-07-09 76      4      3597.90
10253    2014-07-10 34      3      1444.80
10254    2014-07-11 14      5      556.62
10255    2014-07-12 68      9      2490.50
10256    2014-07-15 88      3      517.80
10257    2014-07-16 35      4      1119.90
...
```

SQL Server typically doesn't persist the view's result anywhere; rather it internally keeps the query text and some additional metadata information about the view and its columns in catalog objects. When you query the view, SQL Server expands the view definition and queries the underlying tables. This can be seen in the execution plan of the last query in Figure 3-1.

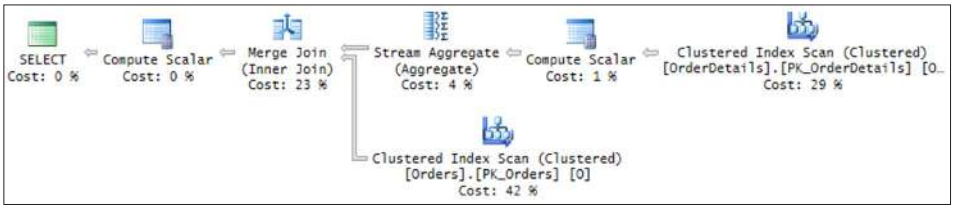


FIGURE 3-1 Execution plan for query against Sales.OrderTotals view

Observe that the plan has no mention of the view; rather it shows that the clustered indexes of the Orders and OrderDetails tables are scanned. You would get the same plan if you issued the following query straight against the underlying tables:

```
SELECT
  O.orderid, O.orderdate, O.custid, O.empid,
  CAST(SUM(OD.qty * OD.unitprice * (1 - OD.discount))
       AS NUMERIC(12, 2)) AS val
FROM Sales.Orders AS O
  INNER JOIN Sales.OrderDetails AS OD
    ON O.orderid = OD.orderid
GROUP BY
  O.orderid, O.custid, O.empid, O.shipperid, O.orderdate,
  O.requireddate, O.shippeddate;
```

If you want to get the definition of an existing view (or other module), use the OBJECT_DEFINITION function, like so:

```
PRINT OBJECT_DEFINITION(OBJECT_ID(N'Sales.OrderTotals'));
```

T-SQL supports defining views based on a query against a CTE. As an example, the following code defines a view called Sales.CustLast5OrderDates based on a query that for each customer, returns the last five distinct order dates:

```
CREATE OR ALTER VIEW Sales.CustLast5OrderDates
  WITH SCHEMABINDING
AS

WITH C AS
(
  SELECT
    custid, orderdate,
    DENSE_RANK() OVER(PARTITION BY custid ORDER BY orderdate DESC) AS pos
  FROM Sales.Orders
)
SELECT custid, [1], [2], [3], [4], [5]
FROM C
  PIVOT(MAX(orderdate) FOR pos IN ([1], [2], [3], [4], [5])) AS P;
GO
```

If you need a refresher of the meaning and syntax of the DENSE_RANK function and PIVOT operator, those were covered in Chapter 2, Skill 2.3. Examine the code and make sure you understand it well. The query that defines the CTE named C returns for each order the customer ID, order date, and dense rank of the order date (descending) for the customer. The outer query against C groups the data implicitly by the customer ID, and pivots the five most recent distinct order dates using the artificial MAX aggregate. The aggregate is artificial in the sense that for each customer and position there's only one distinct order date, but the PIVOT syntax requires you to use an aggregate function to return it.

Notice the use of square brackets to delimit the target column names representing the positions of the order dates. In T-SQL, irregular identifiers such as ones that start with a digit must be delimited. If you remove the delimiters from the columns in the IN clause, you get a syntax error. If you remove them from the columns in the SELECT list, instead of getting the values of the columns [1], [2] and on, which represent order dates, you get back the constants 1, 2, and on. Try it.

Query the view:

```
SELECT custid, [1], [2], [3], [4], [5]
FROM Sales.CustLast5OrderDates;
```

This code generates the following output, shown here in abbreviated form:

```
custid      1          2          3          4          5
----------- ---------- ---------- ---------- ---------- ----------
1           2016-04-09 2016-03-16 2016-01-15 2015-10-13 2015-10-03
2           2016-03-04 2015-11-28 2015-08-08 2014-09-18 NULL
3           2016-01-28 2015-09-25 2015-09-22 2015-06-19 2015-05-13
4           2016-04-10 2016-03-16 2016-03-03 2016-02-02 2015-12-24
5           2016-03-04 2016-02-06 2016-02-03 2016-01-28 2016-01-16
6           2016-04-29 2016-03-17 2016-01-27 2015-07-29 2015-06-27
7           2016-01-12 2015-09-23 2015-08-12 2015-06-30 2015-06-12
8           2016-03-24 2015-12-29 2014-10-10 NULL       NULL
9           2016-05-06 2016-03-11 2016-03-06 2016-02-09 2016-02-05
10          2016-04-24 2016-04-23 2016-04-16 2016-03-27 2016-03-25
...
```

A view can even be defined based on multiple CTEs. Consider the following example, which defines a view called CustTop5OrderValues:

```
CREATE OR ALTER VIEW Sales.CustTop5OrderValues
  WITH SCHEMABINDING
AS

WITH C1 AS
(
  SELECT
    O.orderid, O.custid,
    CAST(SUM(OD.qty * OD.unitprice * (1 - OD.discount))
         AS NUMERIC(12, 2)) AS val
  FROM Sales.Orders AS O
    INNER JOIN Sales.OrderDetails AS OD
      ON O.orderid = OD.orderid
  GROUP BY
    O.orderid, O.custid
),
C2 AS
(
  SELECT
    custid, val,
    ROW_NUMBER() OVER(PARTITION BY custid ORDER BY val DESC, orderid DESC) AS pos
  FROM C1
)
SELECT custid, [1], [2], [3], [4], [5]
FROM C2
  PIVOT(MAX(val) FOR pos IN ([1], [2], [3], [4], [5])) AS P;
GO
```

The code defines a CTE called C1, which computes net order values. The code then defines a CTE called C2, which queries the CTE called C1. It computes row numbers that position orders within each customer partition, ordered by value, descending, and then order ID, descending, as a tiebreaker. The last step is a query against C2 that pivots the five highest order values for each customer to five separate columns named [1], [2], [3], [4] and [5].

Run the following code to query the view:

```
SELECT custid, [1], [2], [3], [4], [5]
FROM Sales.CustTop5OrderValues;
```

This code generates the following output, shown here in abbreviated form:

```
custid  1        2        3        4        5
------- -------- -------- -------- -------- --------
1       933.50   878.00   845.80   814.50   471.20
2       514.40   479.75   320.00   88.80    NULL
3       2082.00  1940.85  813.37   749.06   660.00
4       4441.25  2142.90  1641.00  1477.00  899.00
5       3815.25  3192.65  2222.40  2048.21  1835.70
6       858.00   677.00   625.00   464.00   330.00
7       7390.20  1994.52  1838.20  1761.00  1420.00
8       3026.85  982.00   224.00   NULL     NULL
9       2550.00  2436.18  1979.23  1948.50  1930.40
10      4422.00  3118.00  1892.25  1832.80  1447.50
...
```

As another example for a view that is based on multiple CTEs, the following code creates a view called Sales.OrderValuePcts:

```
CREATE OR ALTER VIEW Sales.OrderValuePcts
  WITH SCHEMABINDING
AS

WITH OrderTotals AS
(
  SELECT
    O.orderid, O.custid,
    SUM(OD.qty * OD.unitprice * (1 - OD.discount)) AS val
  FROM Sales.Orders AS O
    INNER JOIN Sales.OrderDetails AS OD
      ON O.orderid = OD.orderid
  GROUP BY
    O.orderid, O.custid
),
GrandTotal AS
(
  SELECT SUM(val) AS grandtotalval FROM OrderTotals
),
CustomerTotals AS
(
  SELECT custid, SUM(val) AS custtotalval
  FROM OrderTotals
  GROUP BY custid
)
SELECT
  O.orderid, O.custid,
  CAST(O.val AS NUMERIC(12, 2)) AS val,
  CAST(O.val / G.grandtotalval * 100.0 AS NUMERIC(5, 2)) AS pctall,
  CAST(O.val / C.custtotalval * 100.0 AS NUMERIC(5, 2)) AS pctcust
FROM OrderTotals AS O
  CROSS JOIN GrandTotal AS G
  INNER JOIN CustomerTotals AS C
    ON O.custid = C.custid;
GO
```

The OrderTotals CTE computes net order values; the CTE GrandTotal computes the grand total value, and the CTE CustomerTotals computes customer total values. The outer query joins all three CTEs and computes percentages of the order value out of both the grand total and the customer total.

Run the following code to query the view:

```
SELECT orderid, custid, val, pctall, pctcust
FROM Sales.OrderValuePcts;
```

This code generates the following output, shown here in abbreviated form:

```
orderid  custid  val     pctall  pctcust
-------- ------- ------- ------- --------
10835    1       845.80  0.07    19.79
10952    1       471.20  0.04    11.03
10643    1       814.50  0.06    19.06
10692    1       878.00  0.07    20.55
11011    1       933.50  0.07    21.85
10702    1       330.00  0.03    7.72
10625    2       479.75  0.04    34.20
10759    2       320.00  0.03    22.81
10308    2       88.80   0.01    6.33
10926    2       514.40  0.04    36.67
...
```

As a reminder, in Chapter 2 you learned about window functions, which could be used as an alternative to grouping and joining. The same task can be achieved with much less code with window functions, like so:

```
CREATE OR ALTER VIEW Sales.OrderValuePcts
  WITH SCHEMABINDING
AS

WITH OrderTotals AS
(
  SELECT
    O.orderid, O.custid,
    CAST(SUM(OD.qty * OD.unitprice * (1 - OD.discount)) AS NUMERIC(12, 2)) AS val
  FROM Sales.Orders AS O
    INNER JOIN Sales.OrderDetails AS OD
      ON O.orderid = OD.orderid
  GROUP BY
    O.orderid, O.custid
)
SELECT
  orderid, custid, val,
  CAST(val / SUM(val) OVER() * 100.0 AS NUMERIC(5, 2)) AS pctall,
  CAST(val / SUM(val) OVER(PARTITION BY custid) * 100.0 AS NUMERIC(5, 2)) AS pctcust
FROM OrderTotals;
GO
```

Views can also be used to restrict access to only filtered representation of the data. For instance, suppose that you want to grant users from the US branch access to only customers from the US. You can achieve this by creating a view that filters only customers from the US, like so:

```
CREATE OR ALTER VIEW Sales.USACusts
  WITH SCHEMABINDING
AS

SELECT
  custid, companyname, contactname, contacttitle, address,
  city, region, postalcode, country, phone, fax
FROM Sales.Customers
WHERE country = N'USA';
GO
```

You then grant users from the US branch permissions to query the view, and do not grant permissions to access the underlying table directly.

View attributes

As you have noticed, all view definitions in my examples specify the SCHEMABINDING option. This option prevents structural changes to underlying objects while the view exists. To demonstrate what can happen without this option, run the following code to alter the view from the last example in the previous section to not include the SCHEMABINDING option:

```
CREATE OR ALTER VIEW Sales.USACusts
AS

SELECT
  custid, companyname, contactname, contacttitle, address,
  city, region, postalcode, country, phone, fax
FROM Sales.Customers
WHERE country = N'USA';
GO
```

It's important to remember that when you alter a view, either with the ALTER VIEW command, or with the CREATE OR ALTER VIEW command, you have to specify again any view attributes that you want to preserve. The main benefit in altering a view, as opposed to dropping and recreating it, is that permissions are preserved.

Now that our view does not include the SCHEMABINDING option, you're allowed to alter the underlying table definition. Run the following code to drop the address column from the table (in a transaction that you roll back to undo the change):

```
BEGIN TRAN;
  ALTER TABLE Sales.Customers DROP COLUMN address;
ROLLBACK TRAN; -- undo change
```

The code runs successfully.

Alter the view to include the SCHEMABINDING attribute:

```
CREATE OR ALTER VIEW Sales.USACusts
  WITH SCHEMABINDING
AS

SELECT
  custid, companyname, contactname, contacttitle, address,
  city, region, postalcode, country, phone, fax
FROM Sales.Customers
WHERE country = N'USA';
GO
```

Try to drop the column again:

```
ALTER TABLE Sales.Customers DROP COLUMN address;
```

This time the attempt is rejected, and you get the following errors:

```
Msg 5074, Level 16, State 1, Line 247
The object 'USACusts' is dependent on column 'address'.
Msg 4922, Level 16, State 9, Line 247
ALTER TABLE DROP COLUMN address failed because one or more objects access this column.
```

T-SQL supports a view attribute called ENCRYPTION that causes SQL Server to obfuscate the object definition that is stored internally. Without this attribute, you can get the object definition by using the OBJECT_DEFINITION function, like so:

```
SELECT OBJECT_DEFINITION(OBJECT_ID(N'Sales.USACusts'));
```

This code returns the requested view's definition. Alternatively you can use the sp_helptext procedure, or query the sys.syscomments view directly. Next, alter the view definition to include the ENCRYPTION attribute, like so (don't forget to re-specify the SCHEMABINDING option otherwise the view is altered without it):

```
CREATE OR ALTER VIEW Sales.USACusts
  WITH SCHEMABINDING, ENCRYPTION
AS

SELECT
  custid, companyname, contactname, contacttitle, address,
  city, region, postalcode, country, phone, fax
FROM Sales.Customers
WHERE country = N'USA';
GO
```

Run the following code to try and get the object definition:

```
SELECT OBJECT_DEFINITION(OBJECT_ID(N'Sales.USACusts'));
```

This time the code returns a NULL.

Modifying data through views

You're not limited to only issuing SELECT queries against table expressions, such as views, rather, you're also allowed to issue modification statements against those. Because a table expression is a reflection of data from some underlying tables, it's those underlying tables that are affected by the modification. If you're modifying the data through a view, you do need appropriate permissions assigned to you against the view.

To demonstrate this capability, I use the Sales.USACusts view, which you create by running the following code:

```
CREATE OR ALTER VIEW Sales.USACusts
  WITH SCHEMABINDING
AS

SELECT
  custid, companyname, contactname, contacttitle, address,
  city, region, postalcode, country, phone, fax
FROM Sales.Customers
WHERE country = N'USA';
GO
```

Run the following code to add a customer to the Sales.Customers table through the view:

```
INSERT INTO Sales.USACusts(
  companyname, contactname, contacttitle, address,
  city, region, postalcode, country, phone, fax)
VALUES(
  N'Customer AAAAA', N'Contact AAAAA', N'Title AAAAA', N'Address AAAAA',
  N'Redmond', N'WA', N'11111', N'USA', N'111-1111111', N'111-1111111');
```

Query the underlying table to make sure that the new row made it there:

```
SELECT custid, companyname, country
FROM Sales.Customers
WHERE custid = SCOPE_IDENTITY();
```

You get the following output confirming that it did:

```
custid      companyname                              country
----------- ---------------------------------------- ---------------
92          Customer AAAAA                           USA
```

There are some very sensible restrictions on modifications through table expressions. One of them is that if the inner query joins multiple tables, INSERT and UPDATE statements are allowed to affect only one target table at a time. Also, you cannot insert rows through a table expression if it doesn't include at least one column from the underlying table that doesn't somehow get its values automatically (for example by allowing NULLs or having a default value).

Note that normally, you are allowed to insert and update rows through the view even if the modification contradicts the inner query's filter. In our example, you can insert customers from a non-US country, or update an existing customer's country to one other than the US. The effect seems strange because the underlying Sales.Customers table becomes modified, but when you query the view, you won't see the modified rows because they don't satisfy the view's filter anymore.

As an example, run the following code to add a customer from the UK through the view:

```
INSERT INTO Sales.USACusts(
  companyname, contactname, contacttitle, address,
  city, region, postalcode, country, phone, fax)
VALUES(
  N'Customer BBBBB', N'Contact BBBBB', N'Title BBBBB', N'Address BBBBB',
  N'London', NULL, N'22222', N'UK', N'222-2222222', N'222-2222222');
```

The code runs successfully, and the row is added to the underlying table.

Query the view to look for the new customer:

```
SELECT custid, companyname, country
FROM Sales.USACusts
WHERE custid = SCOPE_IDENTITY();
```

You get an empty set back:

```
custid      companyname                              country
----------- ---------------------------------------- ---------------
```

Query the underlying table to look for the new customer:

```
SELECT custid, companyname, country
FROM Sales.Customers
WHERE custid = SCOPE_IDENTITY();
```

This time you do get the new customer row back:

```
custid      companyname                              country
----------- ---------------------------------------- ---------------
93          Customer BBBBB                           UK
```

T-SQL supports an option called CHECK OPTION that prevents inserting or updating rows through the view if the change contradicts the inner query's filter. This option has a somewhat similar effect to a CHECK constraint in a table. You add it at the end of the inner query, like so:

```
CREATE OR ALTER VIEW Sales.USACusts
  WITH SCHEMABINDING
AS

SELECT
  custid, companyname, contactname, contacttitle, address,
  city, region, postalcode, country, phone, fax
FROM Sales.Customers
WHERE country = N'USA'
WITH CHECK OPTION;
GO
```

Run the following code to try again to add a customer from the UK through the view:

```
INSERT INTO Sales.USACusts(
  companyname, contactname, contacttitle, address,
  city, region, postalcode, country, phone, fax)
VALUES(
  N'Customer CCCCC', N'Contact CCCCC', N'Title CCCCC', N'Address CCCCC',
  N'London', NULL, N'33333', N'UK', N'333-3333333', N'333-3333333');
```

This time the code fails and you get the following error:

```
Msg 550, Level 16, State 1, Line 352
The attempted insert or update failed because the target view either specifies WITH
CHECK OPTION or spans a view that specifies WITH CHECK OPTION and one or more rows
resulting from the operation did not qualify under the CHECK OPTION constraint.
The statement has been terminated.
```

Curiously, the CHECK option in a view does differ from a CHECK constraint in a table in how the two handle NULLs, if those are allowed in the target column. For example, suppose that the country column allowed NULLs. A CHECK constraint based on the predicate country = N'USA' would have allowed rows with a NULL country, whereas a view with a filter based on the same predicate and the CHECK option would have regected such rows.

Indexed views

Recall that when you query a view, SQL Server expands the view definition, and optimizes the code against the underlying tables. I demonstrated this earlier. But what if the inner query is quite expensive, and you query the view frequently? You want to avoid the repetition of the work that is involved every time you query the view. To achieve this, you create a clustered index on the view, and this way you persist the view's result within the clustered index B-tree structure. There is an extra cost every time you modify data in the underlying tables because SQL Server needs to modify the indexed view, like it would need to modify other indexes on the tables. So, similar to regular indexes, the tradeoff is faster queries at the cost of extra write cost and space.

To demonstrate the use of indexed views, I use the Sales.OrderTotals view from the previous examples, which you create by running the following code:

```
CREATE OR ALTER VIEW Sales.OrderTotals
  WITH SCHEMABINDING
AS

SELECT
  O.orderid, O.custid, O.empid, O.shipperid,  O.orderdate,
  O.requireddate, O.shippeddate,
  SUM(OD.qty) AS qty,
  CAST(SUM(OD.qty * OD.unitprice * (1 - OD.discount))
       AS NUMERIC(12, 2)) AS val
FROM Sales.Orders AS O
  INNER JOIN Sales.OrderDetails AS OD
    ON O.orderid = OD.orderid
GROUP BY
  O.orderid, O.custid, O.empid, O.shipperid, O.orderdate,
  O.requireddate, O.shippeddate;
GO
```

The first index that you create on the view has to be clustered and unique. Run the following code to try and create such an index on the view:

```
CREATE UNIQUE CLUSTERED INDEX idx_cl_orderid ON Sales.OrderTotals(orderid);
```

As it turns out, there are restrictions and requirements to allow you to create an indexed view. One of them is that the view header has to have the SCHEMABINDING attribute, which in our case is fulfilled. Another is that if the query is a grouped query, it has to include the COUNT_BIG aggregate. SQL Server needs to track the group row counts to know when a group needs to be eliminated as a result of deletes or updates of underlying detail rows. Consequently, your attempt to create the index fails, and you get the following error:

```
Msg 10138, Level 16, State 1, Line 98
Cannot create index on view 'TSQLV4.Sales.OrderTotals' because its select list does not
include a proper use of COUNT_BIG. Consider adding COUNT_BIG(*) to select list.
```

To satisfy this requirement, you alter the view definition and add the COUNT_BIG function by running the following code:

```
CREATE OR ALTER VIEW Sales.OrderTotals
  WITH SCHEMABINDING
AS

SELECT
  O.orderid, O.custid, O.empid, O.shipperid,  O.orderdate,
  O.requireddate, O.shippeddate,
  SUM(OD.qty) AS qty,
  CAST(SUM(OD.qty * OD.unitprice * (1 - OD.discount))
       AS NUMERIC(12, 2)) AS val,
  COUNT_BIG(*) AS numorderlines
FROM Sales.Orders AS O
  INNER JOIN Sales.OrderDetails AS OD
    ON O.orderid = OD.orderid
GROUP BY
  O.orderid, O.custid, O.empid, O.shipperid, O.orderdate,
  O.requireddate, O.shippeddate;
GO
```

Try to create the index again:

```
CREATE UNIQUE CLUSTERED INDEX idx_cl_orderid ON Sales.OrderTotals(orderid);
```

The attempt fails again with the following error:

```
Msg 8668, Level 16, State 0, Line 124
Cannot create the clustered index 'idx_cl_orderid' on view 'TSQLV4.Sales.OrderTotals'
because the select list of the view contains an expression on result of aggregate
function or grouping column. Consider removing expression on result of aggregate
function or grouping column from select list.
```

As it turns out, you're not allowed to manipulate the result of an aggregate calculation, and our query casts the total order value, which is computed with the SUM aggregate, to NUMERIC(12, 2). In order to be able to create the index, you need to remove the manipulation that is applied by the CAST function, like so:

```
CREATE OR ALTER VIEW Sales.OrderTotals
  WITH SCHEMABINDING
AS

SELECT
  O.orderid, O.custid, O.empid, O.shipperid,  O.orderdate,
  O.requireddate, O.shippeddate,
  SUM(OD.qty) AS qty,
  SUM(OD.qty * OD.unitprice * (1 - OD.discount)) AS val,
  COUNT_BIG(*) AS numorderlines
FROM Sales.Orders AS O
  INNER JOIN Sales.OrderDetails AS OD
    ON O.orderid = OD.orderid
GROUP BY
  O.orderid, O.custid, O.empid, O.shipperid, O.orderdate,
  O.requireddate, O.shippeddate;
GO
```

Try again to create the index:

```
CREATE UNIQUE CLUSTERED INDEX idx_cl_orderid ON Sales.OrderTotals(orderid);
```

This time the index is created successfully.

To recap, the view has to include SCHEMABINDING, the first index has to be clustered and unique, if the query is a grouped query it has to include the COUNT_BIG aggregate functions, and you're not allowed to manipulate the result of aggregate functions. You can find a more complete list of requirements and restriction at *https://msdn.microsoft.com/en-us/library/ms191432.aspx*.

Once you successfully created a clustered index on a view, you're allowed to create additional nonclustered indexes. Run the following code to create a number of nonclustered indexes on the Sales.OrderTotal view:

```
CREATE NONCLUSTERED INDEX idx_nc_custid      ON Sales.OrderTotals(custid);
CREATE NONCLUSTERED INDEX idx_nc_empid       ON Sales.OrderTotals(empid);
CREATE NONCLUSTERED INDEX idx_nc_shipperid   ON Sales.OrderTotals(shipperid);
CREATE NONCLUSTERED INDEX idx_nc_orderdate   ON Sales.OrderTotals(orderdate);
CREATE NONCLUSTERED INDEX idx_nc_shippeddate ON Sales.OrderTotals(shippeddate);
```

Run the following code to query the view:

```
SELECT orderid, custid, empid, shipperid, orderdate,
  requireddate, shippeddate, qty, val, numorderlines
FROM Sales.OrderTotals;
```

As long as you're running the code on an Enterprise or Developer edition of SQL Server, the optimizer considers using indexes on the view without any special instructions. This can be seen in the execution plan of the last query, as shown in Figure 3-2.

FIGURE 3-2 Query plan with index on OrderTotals view

If you're using a non-Enterprise or Developer edition of SQL Server, you need to indicate the NOEXPAND hint against the view in order for SQL Server to not expand the view definition, but rather consider using the index on the view. The following query demonstrates using this hint:

```
SELECT orderid, custid, empid, shipperid, orderdate,
  requireddate, shippeddate, qty, val, numorderlines
FROM Sales.OrderTotals WITH (NOEXPAND);
```

Curiously, as long as you do use Enterprise or Developer edition, SQL Server considers using the indexes on the view even when you query the underlying tables. For example, consider the following query.

```
SELECT
  O.orderid, O.custid, O.empid, O.shipperid,  O.orderdate,
  O.requireddate, O.shippeddate,
  SUM(OD.qty) AS qty,
  CAST(SUM(OD.qty * OD.unitprice * (1 - OD.discount))
       AS NUMERIC(12, 2)) AS val
FROM Sales.Orders AS O
  INNER JOIN Sales.OrderDetails AS OD
    ON O.orderid = OD.orderid
GROUP BY
  O.orderid, O.custid, O.empid, O.shipperid, O.orderdate,
  O.requireddate, O.shippeddate;
```

The plan for this query is shown in Figure 3-3.

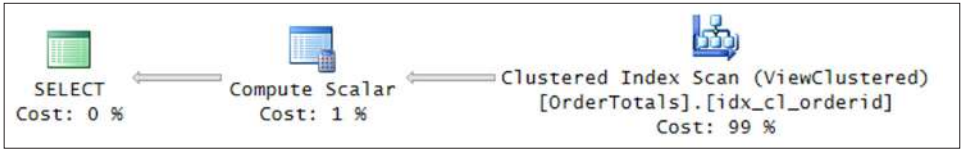


FIGURE 3-3 Another query plan with index on OrderTotals view

Recall that you had to remove the CAST function that you originally applied to the SUM aggregate that computed the net order value in order to create the indexed view. If you want the user to be able to query a simplified view with the casted value, you could create an intermediate indexed view without the CAST expression, and then a non-indexed view with the original name that adds the CAST expression, like so:

```
-- create intermediate view
CREATE OR ALTER VIEW Sales.VOrderTotals
  WITH SCHEMABINDING
AS

SELECT
  O.orderid, O.custid, O.empid, O.shipperid,  O.orderdate,
  O.requireddate, O.shippeddate,
  SUM(OD.qty) AS qty,
  SUM(OD.qty * OD.unitprice * (1 - OD.discount)) AS val,
  COUNT_BIG(*) AS numorderlines
FROM Sales.Orders AS O
  INNER JOIN Sales.OrderDetails AS OD
    ON O.orderid = OD.orderid
GROUP BY
  O.orderid, O.custid, O.empid, O.shipperid, O.orderdate,
  O.requireddate, O.shippeddate;
GO

-- create indexes on view
CREATE UNIQUE CLUSTERED INDEX idx_cl_orderid ON Sales.VOrderTotals(orderid);
CREATE NONCLUSTERED INDEX idx_nc_custid      ON Sales.VOrderTotals(custid);
CREATE NONCLUSTERED INDEX idx_nc_empid       ON Sales.VOrderTotals(empid);
CREATE NONCLUSTERED INDEX idx_nc_shipperid   ON Sales.VOrderTotals(shipperid);
CREATE NONCLUSTERED INDEX idx_nc_orderdate   ON Sales.VOrderTotals(orderdate);
```

```
CREATE NONCLUSTERED INDEX idx_nc_shippeddate ON Sales.VOrderTotals(shippeddate);

-- create view with CAST
CREATE OR ALTER VIEW Sales.OrderTotals
  WITH SCHEMABINDING
AS

SELECT
  orderid, custid, empid, shipperid,  orderdate, requireddate, shippeddate, qty,
  CAST(val AS NUMERIC(12, 2)) AS val
FROM Sales.VOrderTotals;
GO
```

Run the following code to query the view:

```
SELECT orderid, custid, empid, shipperid,  orderdate,
  requireddate, shippeddate, qty, val
FROM Sales.OrderTotals;
```

The plan for this query is shown in Figure 3-4.

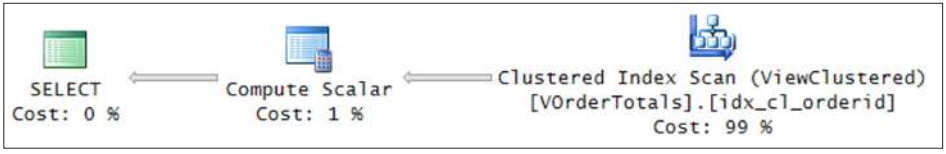


FIGURE 3-4 Query plan with index on VOrderTotals view

Observe that the clustered index on the VOrderTotals view is used.

When you're done, run the following code for cleanup:

```
DROP VIEW IF EXISTS
  Sales.OrderTotals, Sales.VOrderTotals, Sales.CustLast5OrderDates,
  Sales.CustTop5OrderValues, Sales.OrderValuePcts, Sales.USACusts;
```

User-defined functions

A user-defined function (UDF) is a routine that accepts parameters, applies calculations, and returns either a scalar-valued or a table-valued result. SQL Server supports developing functions using either T-SQL, or the common language runtime (CLR). The focus of the exam—and hence the book's—is the T-SQL kind.

A user-defined function can appear in places in your code where a scalar-valued or table-valued expression can appear, such as in a query, a computed column, and a CHECK constraint. It can even appear in the definition of another user-defined function. You can use a user-defined function to replace a stored procedure when you want to be able to query its result. You can use it as an alternative to a view with parameters, since views don't support parameters. You can even use a user-defined function to query an indexed view, and with parameter support, improve its functionality. You can also use a user-defined function to define a filter for a security policy as part of an implementation of row level security.

There are a number of restrictions and limitations on user-defined functions. Within user-define functions you cannot:

- Use error handling
- Modify data (other than in table variables)
- Use data definition language (DDL)
- Use temporary tables
- Use dynamic SQL

You can find a more complete list of requirements and restrictions at *https://msdn.microsoft.com/en-us/library/ms191320.aspx*.

Some of the examples in this section use a table called dbo.Employees (not to be confused with HR.Employees from the sample database). Run the following code to create this table and populate it with sample data:

```
SET NOCOUNT ON;
USE TSQLV4;
DROP TABLE IF EXISTS dbo.Employees;
GO
CREATE TABLE dbo.Employees
(
  empid   INT         NOT NULL CONSTRAINT PK_Employees PRIMARY KEY,
  mgrid   INT         NULL
    CONSTRAINT FK_Employees_Employees REFERENCES dbo.Employees,
  empname VARCHAR(25) NOT NULL,
  salary  MONEY       NOT NULL,
  CHECK (empid <> mgrid)
);

INSERT INTO dbo.Employees(empid, mgrid, empname, salary)
  VALUES(1, NULL, 'David', $10000.00),
        (2, 1, 'Eitan', $7000.00),
        (3, 1, 'Ina', $7500.00),
        (4, 2, 'Seraph', $5000.00),
        (5, 2, 'Jiru', $5500.00),
        (6, 2, 'Steve', $4500.00),
        (7, 3, 'Aaron', $5000.00),
        (8, 5, 'Lilach', $3500.00),
        (9, 7, 'Rita', $3000.00),
        (10, 5, 'Sean', $3000.00),
        (11, 7, 'Gabriel', $3000.00),
        (12, 9, 'Emilia' , $2000.00),
        (13, 9, 'Michael', $2000.00),
        (14, 9, 'Didi', $1500.00);

CREATE UNIQUE INDEX idx_unc_mgr_emp_i_name_sal ON dbo.Employees(mgrid, empid)
  INCLUDE(empname, salary);
```

T-SQL supports three kinds of user-defined functions: *scalar*, *inline table valued* and *multi-statement table valued*. The upcoming sections cover these three types.

Scalar user-defined functions

A scalar user-defined function accepts parameters, applies calculations, and returns a single value. It has a body with flow that can have multiple statements, including queries, and eventually it must invoke a RETURN clause to return the result value. The header of the function defines the input parameters as well as the return type using a RETURNS clause.

As an example, the following code defines a function called dbo.SubtreeTotalSalaries:

```
CREATE OR ALTER FUNCTION dbo.SubtreeTotalSalaries(@mgr AS INT)
  RETURNS MONEY
WITH SCHEMABINDING
AS
BEGIN
  DECLARE @totalsalary AS MONEY;

  WITH EmpsCTE AS
  (
    SELECT empid, salary
    FROM dbo.Employees
    WHERE empid = @mgr

    UNION ALL

    SELECT S.empid, S.salary
    FROM EmpsCTE AS M
      INNER JOIN dbo.Employees AS S
        ON S.mgrid = M.empid
  )
  SELECT @totalsalary = SUM(salary)
  FROM EmpsCTE;

  RETURN @totalsalary;
END;
GO
```

The header of the function defines an input parameter called @mgr representing an input manager ID, and MONEY as the returned type. It specifies the SCHEMABINDING option to prevent structural changes to dependent objects (the Employees table in our case). The body of the function resides between the mandatory BEGIN and END clauses. The code in the function's body declares a local variable called @totalsalary. It then uses a recursive query that identifies the input manager's subordinates (direct and indirect). The outer statement of the CTE then assigns the total salaries of the input manager and all identified subordinates to the variable. Finally, the code returns the value stored in @totalsalary.

EXAM TIP

Remember that during the exam you don't have access to any online or offline resources that can help you figure out the syntax of T-SQL commands. Make sure to memorize the syntax for defining the different types of user-defined functions.

Use the following code to test the function, asking for the total salaries of the subtree of manager 8 (Lilach):

```
SELECT dbo.SubtreeTotalSalaries(8) AS subtreetotal;
```

This code returns the following output:

```
Subtreetotal
--------------------
3500.00
```

With most object types, T-SQL allows you to omit the schema name when referring to the object, in which case it uses implicit schema name resolution. With scalar UDFs, you must use the two-part name including the schema. For instance, try calling the function again, but this time without the schema:

```
SELECT SubtreeTotalSalaries(8) AS subtreetotal;
```

You get the following error:

```
Msg 195, Level 15, State 10, Line 553
'SubtreeTotalSalaries' is not a recognized built-in function name.
```

SQL Server looks for a built-in function with the specified name and since it can't find one, it fails.

As mentioned, you can invoke a user-defined function as part of a query. For example, the following code queries the Employees table, and invokes the function for each employee to return the total salaries of the respective employee's subtree:

```
SELECT empid, mgrid, empname, salary,
  dbo.SubtreeTotalSalaries(empid) AS subtreetotal
FROM dbo.Employees;
```

This code generates the following output:

```
empid  mgrid  empname  salary    subtreetotal
------ ------ -------- --------- -------------
1      NULL   David    10000.00  62500.00
2      1      Eitan    7000.00   28500.00
3      1      Ina      7500.00   24000.00
4      2      Seraph   5000.00   5000.00
5      2      Jiru     5500.00   12000.00
6      2      Steve    4500.00   4500.00
7      3      Aaron    5000.00   16500.00
8      5      Lilach   3500.00   3500.00
10     5      Sean     3000.00   3000.00
9      7      Rita     3000.00   8500.00
11     7      Gabriel  3000.00   3000.00
12     9      Emilia   2000.00   2000.00
13     9      Michael  2000.00   2000.00
14     9      Didi     1500.00   1500.00
```

EXAM TIP

The input and output parameters of functions are not limited to simple types like INT, DATE, and MONEY. They can be more complex types like XML, HIERARCHYID, GEOMETRY, GEOGRAPHY, and user-defined CLR types. Also, input parameters can be assigned with a default, as in @p AS INT = 0. However, when you invoke the function, if you wish to rely on the default value, you must specify the keyword DEFAULT instead of passing a value. You cannot just omit the parameter like you do with stored procedures.

There is an interesting difference between invoking a built-in nondeterministic function like SYSDATETIME directly in a query, and invoking it indirectly from a user-defined function that you then invoke in a query. Recall from Skill 1.3 in Chapter 1 that a *deterministic* function is guaranteed to return the same output across calls given the same inputs, and a *nondeterministic* function isn't. When you invoke a nondeterministic function directly in a query, SQL Server executes it only once for the entire query. The NEWID function is an exception to this rule because SQL Server executes it once per row. When you invoke a nondeterministic built-in function indirectly, from within a user-defined function, and then invoke the user-defined function in a query, the function gets executed once per row.

As an example, the following code queries the Sales.Orders table, and invokes the functions SYSDATETIME (returns the current date and time), RAND without a seed (returns a random float value in the range 0 through 1), and NEWID (returns a globally unique identifier) in the SELECT list:

```
SELECT orderid, SYSDATETIME() AS [SYSDATETIME], RAND() AS [RAND], NEWID() AS [NEWID]
FROM Sales.Orders;
```

This code generated the following output on my system (your result naturally differs):

```
orderid  SYSDATETIME                 RAND               NEWID
-------- -------------------------- ------------------ ------------------------------
-----
11008    2016-11-23 09:41:20.0517170 0.115119415679133  DF0ED3F-53B9-4034-9D4D-
ABF33F3625F0
11019    2016-11-23 09:41:20.0517170 0.115119415679133  F3AB18AE-E897-4714-B7E5-
6962E26237C0
11039    2016-11-23 09:41:20.0517170 0.115119415679133  1D0D899C-8477-4617-B7C9-
797554059AA4
11040    2016-11-23 09:41:20.0517170 0.115119415679133  AF6386D7-122F-43E6-90EF-
AB432A9C03EB
11045    2016-11-23 09:41:20.0517170 0.115119415679133  EC3857B7-854A-4121-8CD5-
74B5B36DB279
...
11050    2016-11-23 09:41:20.0517170 0.115119415679133  9CAA81D2-4A3A-4B36-A8C8-
A1F92D78FC66
11055    2016-11-23 09:41:20.0517170 0.115119415679133  58E95547-DA18-4EAE-BC5C-
09252F266510
11063    2016-11-23 09:41:20.0517170 0.115119415679133  138CE9F6-7FFE-41E0-A507-
4190183CC203
11067    2016-11-23 09:41:20.0517170 0.115119415679133  CB864AFA-0D44-4435-90DA-
A718F200C85D
11069    2016-11-23 09:41:20.0517170 0.115119415679133  CC8B555C-DD9F-45C0-A2AE-
4E468ECED251
```

Observe that both SYSDATETIME and RAND return the same result in all rows even though they are nondeterministic functions. As mentioned, most nondeterministic built-in functions are invoked once per query. Also observe that NEWID behaves differently; it is invoked once per row.

T-SQL supports invoking nondeterministic built-in functions within user-defined functions, as long as they don't have any side effects on the system. The functions SYSDATETIME, RAND (without an input seed), and NEWID are all nondeterministic functions. The SYSDATETIME function doesn't have any side effects on the system and therefore is allowed in user-defined functions. As an example, run the following code to create the user-defined function MySYSDATETIME:

```
CREATE OR ALTER FUNCTION dbo.MySYSDATETIME() RETURNS DATETIME2
AS
BEGIN
  RETURN SYSDATETIME();
END;
GO
```

The function is created successfully.

The NEWID and RAND functions both have side effects in the sense that one function call leaves a mark behind that affects a subsequent function call. Consequently, you're not allowed to invoke NEWID and RAND within user-defined functions. As an example, run the following code in attempt to create a user-defined function called MyRand that calls the built-in RAND function:

```
CREATE OR ALTER FUNCTION dbo.MyRAND() RETURNS FLOAT
AS
BEGIN
  RETURN RAND();
END;
GO
```

The attempt to create the function fails with the following error:

```
Msg 443, Level 16, State 1, Procedure MyRAND, Line 4 [Batch Start Line 516]
Invalid use of a side-effecting operator 'rand' within a function.
```

Curiously, if you invoke the built-in side-effecting function from a view, SQL Server does allow you to query the view from within a user-defined function. This allows you to circumvent the aforementioned restriction. As an example, run the following code to create a view called VRAND that invokes the built-in function RAND and returns the result as a column called myrand:

```
CREATE OR ALTER VIEW dbo.VRAND
AS

SELECT RAND() AS myrand;
GO
```

Next, run the following code to create the function MyRAND, which returns the result of a query against the view:

```
CREATE OR ALTER FUNCTION dbo.MyRAND() RETURNS FLOAT
AS
BEGIN
  RETURN (SELECT myrand FROM dbo.VRAND);
END;
GO
```

Run the following code to query the Sales.Orders table and invoke the functions MySYSDATETIME and MyRAND:

```
SELECT orderid, dbo.MySYSDATETIME() AS mysysdatetime, dbo.MyRAND() AS myrand
FROM Sales.Orders;
```

This code generates the following output:

```
orderid     mysysdatetime               myrand
----------- --------------------------- ----------------------
11008       2016-11-23 09:47:46.5503588 0.0641387937377291
11019       2016-11-23 09:47:46.5518609 0.397630436473708
11039       2016-11-23 09:47:46.5518609 0.906453593195041
11040       2016-11-23 09:47:46.5518609 0.50133925878664
11045       2016-11-23 09:47:46.5518609 0.867531671551516
...
11050       2016-11-23 09:47:46.5634261 0.666663850667723
11055       2016-11-23 09:47:46.5634261 0.452951660618415
11063       2016-11-23 09:47:46.5634261 0.286359018535061
11067       2016-11-23 09:47:46.5634261 0.265970755632963
11069       2016-11-23 09:47:46.5634261 0.453168331889983
```

Observe that SQL Server executed the user-defined functions once per row, unlike earlier when you invoked the built-in functions directly, in which case SQL Server executed the functions only once for the entire query.

Note that in order to use a user-defined function in a persisted computed column or in an indexed view, the function needs to be deterministic. What this means is that the user-defined function must not call nondeterministic functions, plus you need to define it with the SCHEMABINDING attribute. As an example, run the following code to create the function ENDOFYEAR, which accepts a date as input, and returns the corresponding end of year date:

```
CREATE OR ALTER FUNCTION dbo.ENDOFYEAR(@dt AS DATE) RETURNS DATE
AS
BEGIN
  RETURN DATEFROMPARTS(YEAR(@dt), 12, 31);
END;
GO
```

Notice that the function's header does not include the SCHEMABINDING attribute. As such, the user-defined function isn't guaranteed to be deterministic. Next, run the following

code in attempt to create a table with a persistent computed column that is based on the function that you just created:

```
DROP TABLE IF EXISTS dbo.T1;
GO
CREATE TABLE dbo.T1
(
  keycol INT NOT NULL IDENTITY CONSTRAINT PK_T1 PRIMARY KEY,
  dt DATE NOT NULL,
  dtendofyear AS dbo.ENDOFYEAR(dt) PERSISTED
);
```

SQL Server rejects the attempt with the following error:

```
Msg 4936, Level 16, State 1, Line 574
Computed column 'dtendofyear' in table 'T1' cannot be persisted because the column is
non-deterministic.
```

Run the following code to recreate the function, only this time with the SCHEMABINDING attribute:

```
CREATE OR ALTER FUNCTION dbo.ENDOFYEAR(@dt AS DATE)
  RETURNS DATE
WITH SCHEMABINDING
AS
BEGIN
  RETURN DATEFROMPARTS(YEAR(@dt), 12, 31);
END;
GO
```

Run the following code in attempt to create the table again:

```
CREATE TABLE dbo.T1
(
  keycol INT NOT NULL IDENTITY CONSTRAINT PK_T1 PRIMARY KEY,
  dt DATE NOT NULL,
  dtendofyear AS dbo.ENDOFYEAR(dt) PERSISTED
);
```

This time the code completes successfully.

Inline table-valued user-defined functions

An inline table-valued user-defined function is very similar in concept to a view in the sense that it's based on a single query, and you interact with it like a table expression, only unlike a view, it supports input parameters. So you could think of such a function as a parameterized view.

The reason that it's called an *inline* function is because SQL Server inlines, or expands, the inner query definition, and constructs an internal query directly against the underlying tables. The inlining process includes replacing the inner query's references to the input parameters with the passed constants (what's known as *parameter embedding*).

As an example, run the following code to create an inline function called GetPage:

```
CREATE OR ALTER FUNCTION dbo.GetPage(@pagenum AS BIGINT, @pagesize AS BIGINT)
  RETURNS TABLE
WITH SCHEMABINDING
AS
RETURN
  WITH C AS
  (
    SELECT ROW_NUMBER() OVER(ORDER BY orderdate, orderid) AS rownum,
      orderid, orderdate, custid, empid
    FROM Sales.Orders
  )
  SELECT rownum, orderid, orderdate, custid, empid
  FROM C
  WHERE rownum BETWEEN (@pagenum - 1) * @pagesize + 1 AND @pagenum * @pagesize;
GO
```

Notice that the function's header defines the input parameters and says that it returns a table result (RETURN TABLE). There's no body to the function (no BEGIN-END block), rather just a RETURN clause with a query.

This function implements a paging solution where you pass a page number and page size, and the function returns orders from the desired page. The code defines a CTE called C based on a query that computes row numbers for orders, based on orderdate and orderid ordering. The outer query then filters only the rows from C with the row numbers that fall in the requested page. The starting row number is based on the expression (@pagenum - 1) * @pagesize + 1, and the ending row number is based on the expression pagenum * @pagesize. For example, with a page number 3, and a page size of 12, the first row number is 25, and the last row number is 36.

Run the following query to test the function, asking for page 3 with a page size of 12:

```
SELECT rownum, orderid, orderdate, custid, empid
FROM dbo.GetPage(3, 12) AS T;
```

This query generates the following output:

```
rownum  orderid  orderdate  custid  empid
------- -------- ---------- ------- ------
25      10272    2014-08-02 65      6
26      10273    2014-08-05 63      3
27      10274    2014-08-06 85      6
28      10275    2014-08-07 49      1
29      10276    2014-08-08 80      8
30      10277    2014-08-09 52      2
31      10278    2014-08-12 5       8
32      10279    2014-08-13 44      8
33      10280    2014-08-14 5       2
```

```
34      10281    2014-08-14 69      4
35      10282    2014-08-15 69      4
36      10283    2014-08-16 46      3
```

Recall from Chapter 1 that T-SQL supports a built-in feature for ad-hoc paging in the form of a filter called OFFSET-FETCH. With this filter, you can simplify the solution for the task at hand by using the following function definition instead:

```
CREATE OR ALTER FUNCTION dbo.GetPage(@pagenum AS BIGINT, @pagesize AS BIGINT)
  RETURNS TABLE
WITH SCHEMABINDING
AS
RETURN
  SELECT ROW_NUMBER() OVER(ORDER BY orderdate, orderid) AS rownum,
    orderid, orderdate, custid, empid
  FROM Sales.Orders
  ORDER BY orderdate, orderid
  OFFSET (@pagenum - 1) * @pagesize ROWS FETCH NEXT @pagesize ROWS ONLY;
GO
```

With this solution you don't need to use a CTE that defines row numbers and then an outer query that handles the filter, you can handle both tasks in one query. Run the following code to test the altered function:

```
SELECT rownum, orderid, orderdate, custid, empid
FROM dbo.GetPage(3, 12) AS T;
```

You get the same result as before.

As a more complex example for an inline table-valued user-defined function, suppose that you need to implement a function called GetSubtree that accepts a manager ID as input (call it @mgr), and an optional parameter to limit the number of levels (call it @maxlevels); the function should return the set of all subordinates of the input manager, applying the level limit if one was specified. The function should return a NULL as the manager ID of the input manager.

Run the following code to implement the GetSubtree function as an inline table-valued one:

```
DROP FUNCTION IF EXISTS dbo.GetSubtree;
GO
CREATE FUNCTION dbo.GetSubtree(@mgr AS INT, @maxlevels AS INT = NULL)
  RETURNS TABLE
WITH SCHEMABINDING
AS
RETURN
  WITH EmpsCTE AS
  (
    SELECT empid, CAST(NULL AS INT) AS mgrid, empname, salary, 0 as lvl,
      CAST('.' AS VARCHAR(900)) AS sortpath
    FROM dbo.Employees
    WHERE empid = @mgr
```

```
    UNION ALL

    SELECT S.empid, S.mgrid, S.empname, S.salary, M.lvl + 1 AS lvl,
      CAST(M.sortpath + CAST(S.empid AS VARCHAR(10)) + '.' AS VARCHAR(900)) AS sortpath
    FROM EmpsCTE AS M
      INNER JOIN dbo.Employees AS S
        ON S.mgrid = M.empid
        AND (M.lvl < @maxlevels OR @maxlevels IS NULL)
  )
  SELECT empid, mgrid, empname, salary, lvl, sortpath
  FROM EmpsCTE;
```

The function uses a recursive CTE to handle the task. The anchor member returns the row for the input manager, along with the constant manager ID NULL, the level 0, and a sort path that is made of a separator plus the current employee ID plus another separator.

The recursive member joins the managers from the previous round (identified with the recursive reference to the CTE name) with the Employees table to return direct subordinates, provided the level (depth) did not exceed the specified maximum number of levels, if one was specified. The subordinate's level is computed as the corresponding manager's level plus 1, and the subordinate's sort path is computed by concatenating the manager's path with the current employee ID plus a separator.

Note that corresponding columns in the anchor and the recursive queries must have identical types, including length and precision. Since the manager ID of the input manager is supposed to be set to the constant NULL, and the manager IDs of the subordinates are to be integer typed values, you need to explicitly convert the NULL constant to INT in the anchor query. Similarly, since the column sortpath in both the anchor and recursive queries have to have the same type and length, the code explicitly converts the values in both queries to VARCHAR(900).

The following example demonstrates using the function to return the subtree of manager 3, without limiting the number of levels:

```
SELECT empid, REPLICATE(' | ', lvl) + empname AS emp,
  mgrid, salary, lvl, sortpath
FROM dbo.GetSubtree(3, NULL) AS T
ORDER BY sortpath;
```

This code generates the following output:

```
empid  empname           mgrid  salary   lvl  sortpath
------ ----------------- ------ -------- ---- -----------
3      Ina               NULL   7500.00  0    .
7       | Aaron          3      5000.00  1    .7.
11      |  | Gabriel     7      3000.00  2    .7.11.
9       |  | Rita        7      3000.00  2    .7.9.
12      |  |  | Emilia   9      2000.00  3    .7.9.12.
13      |  |  | Michael  9      2000.00  3    .7.9.13.
14      |  |  | Didi     9      1500.00  3    .7.9.14.
```

Ordering the result by the sortpath column guarantees that the presentation order of the result reflects *topological sort order*, meaning that a manager always shows up before his or her subordinates. Using the REPLICATE function the code replicates a string lvl times to achieve a visual indentation effect that reflects the level of the current node with respect to the root of the subtree. It's pretty!

> ***MORE INFO*** **ROW-LEVEL SECURITY**
>
> **You can also use inline table-valued user-defined functions as a filter predicate for a security policy as part of a solution for row-level security. You can find the details at *https://msdn.microsoft.com/en-us/library/dn765131.aspx*.**

Multistatement table-valued user-defined functions

A multistatement table-valued user-defined function is a table function, so from the user's perspective, it's used as a source table in a query much like an inline table-valued function is used. However, instead of being based on a single query, the multistatement function declares a returned table variable in its header, and then its body is responsible for filling the returned table variable with rows. Whenever you query the function, behind the scenes SQL Server creates the table variable that is defined in the function's header, runs the flow in the function's body to fill it with rows, and then as soon as the function executes the RETURN command, SQL Server hands the table variable to the calling query.

As an example, run the following code to create an alternative implementation of the GetSubtree function, this time as a multistatement table-valued function:

```
DROP FUNCTION IF EXISTS dbo.GetSubtree;
-- cannot use CREATE OR ALTER to change the function type
GO
CREATE FUNCTION dbo.GetSubtree (@mgrid AS INT, @maxlevels AS INT = NULL)
RETURNS @Tree TABLE
(
  empid    INT          NOT NULL PRIMARY KEY,
  mgrid    INT          NULL,
  empname  VARCHAR(25)  NOT NULL,
  salary   MONEY        NOT NULL,
  lvl      INT          NOT NULL,
  sortpath VARCHAR(892) NOT NULL ,
  INDEX idx_lvl_empid_sortpath NONCLUSTERED(lvl, empid, sortpath)
)
WITH SCHEMABINDING
AS
BEGIN
  DECLARE @lvl AS INT = 0;

  -- insert subtree root node into @Tree
  INSERT INTO @Tree(empid, mgrid, empname, salary, lvl, sortpath)
    SELECT empid, NULL AS mgrid, empname, salary, @lvl AS lvl, '.' AS sortpath
    FROM dbo.Employees
    WHERE empid = @mgrid;
```

```
  WHILE @@ROWCOUNT > 0 AND (@lvl < @maxlevels OR @maxlevels IS NULL)
  BEGIN
    SET @lvl += 1;

    -- insert children of nodes from prev level into @Tree
    INSERT INTO @Tree(empid, mgrid, empname, salary, lvl, sortpath)
      SELECT S.empid, S.mgrid, S.empname, S.salary, @lvl AS lvl,
        M.sortpath + CAST(S.empid AS VARCHAR(10)) + '.' AS sortpath
      FROM dbo.Employees AS S
        INNER JOIN @Tree AS M
          ON S.mgrid = M.empid AND M.lvl = @lvl - 1;
  END;

  RETURN;
END;
GO
```

The header of the function defines a table variable called @Tree as the returned value. Then the code in the body of the function (observe the BEGIN-END block) starts by declaring a level counter called @lvl and initializes it with 0. The code continues to insert the row for the input manager into the table variable along with the just initialized level and initial sort path. The code then defines a loop that keeps iterating as long as the number of rows affected by the previous insert is nonzero, and we haven't yet reached the maximum number of levels if one was specified. In each round, the loop increments the level counter and inserts the next level of subordinates into the table variable. Finally, the code executes the RETURN command, causing the function to exit and hand the table variable to the calling query.

Use the following code to test the function:

```
SELECT empid, REPLICATE(' | ', lvl) + empname AS emp,
  mgrid, salary, lvl, sortpath
FROM dbo.GetSubtree(3, NULL) AS T
ORDER BY sortpath;
```

This code generates the following output:

```
empid  empname           mgrid  salary   lvl  sortpath
------ ----------------- ------ -------- ---- -----------
3      Ina               NULL   7500.00  0    .
7       | Aaron           3      5000.00  1    .7.
11      |  | Gabriel      7      3000.00  2    .7.11.
9       |  | Rita         7      3000.00  2    .7.9.
12      |  |  | Emilia    9      2000.00  3    .7.9.12.
13      |  |  | Michael   9      2000.00  3    .7.9.13.
14      |  |  | Didi      9      1500.00  3    .7.9.14.
```

If you're wondering why bother implementing a task as a multistatement function instead of an inline one, there could be a number of reasons. The recursive query in the inline function uses a spool (worktable) to store the intermediate results, but you have no control over the indexing on that spool. With the multistatement function, you control the definition of the table variable, including indexing. Furthermore, the inline function is only allowed to return a query, whereas the multistatement function can have complete flow with procedural

logic. It just so happens that the task in our specific example can be handled with both kinds of functions, so you can try them both, and compare their performance to see which one works better for you. But some tasks cannot be handled with a single query, rather require a multi-step procedural solution.

When you're done, run the following code for cleanup:

```
DROP TABLE IF EXISTS dbo.T1;

DROP VIEW IF EXISTS dbo.VRAND;

DROP FUNCTION IF EXISTS dbo.MySYSDATETIME, dbo.MyRAND, dbo.ENDOFYEAR,
  dbo.SubtreeTotalSalaries, dbo.GetPage, dbo.GetSubtree;

DROP TABLE IF EXISTS dbo.Employees;
```

Stored procedures

Stored procedures are routines that support flow with multiple steps. They support many T-SQL elements that user-defined functions don't, like modifying data in the database and even applying data definition changes to database objects (DDL), using temporary tables, dynamic SQL, error handling, and more. These are clear advantages compared to user-defined functions; however, unlike with functions, stored procedures cannot be embedded in queries.

Stored procedures support input and output parameters and can also return result sets of queries. SQL Server caches the execution plans of the stored procedure's queries, and typically reuses them in subsequent executions of the procedure to save the time, CPU and memory resources that are associated with optimizing the queries.

Stored procedures provide many benefits compared to implementing the business logic in the application. They encapsulate the logic to allow reusability and hiding of complexity. It's much easier to apply changes to a stored procedure with a simple ALTER PROC command compared to deploying changes in the application. Also, with stored procedures you tend to have less network traffic because when you call a stored procedure from the application, all that is passed through the network is just the procedure name and its parameters. The flow runs in the database engine, and then only the final result is sent to the application. When you implement the logic in the application, you usually get more roundtrips between the application and the database, and consequently more network traffic.

Stored procedures also simplify handling security in the database. Often you don't want to grant users with permissions to directly query and modify data in tables, rather you want them to be able to achieve such tasks only indirectly through stored procedures. To achieve this, grant the users with EXECUTE permissions on the stored procedure while not granting them direct access to underlying objects.

EXAM TIP

The exam expects you to know what the conditions are in which each type of object is appropriate and supported. Make sure you understand the conditions when it's appropriate to use views, user-defined functions of the various types, and stored procedures.

The upcoming sections cover working with stored procedures, using dynamic SQL within procedures, using output parameters and modifying data, and also using cursors within stored procedures. Later in the chapter, Skill 3.2 covers error handling within stored procedures.

Working with stored procedures

A stored procedure is a reusable routine that supports input and output parameters, and even returning result sets of queries. This section starts with a simple example for using input parameters and returning a result set of a query. I demonstrate using output parameters in a later section.

Suppose that you are given a task to create a stored procedure that handles what's called *dynamic search conditions* to query and filter data from the Sales.Orders table. The procedure should have four optional input parameters for filtering the orders by order ID, order date, customer ID, and employee ID. The executing user determines which combination of columns to filter by. The way you make the input parameters optional is by defining them with a default NULL. If the user doesn't specify a value for an input parameter, it is set to the default NULL, telling you that you're not supposed to apply a filter to the corresponding column.

Run the following code to create the GetOrders stored procedure, which handles the task at hand:

```
CREATE OR ALTER PROC dbo.GetOrders
  @orderid   AS INT  = NULL,
  @orderdate AS DATE = NULL,
  @custid    AS INT  = NULL,
  @empid     AS INT  = NULL
AS

SET XACT_ABORT, NOCOUNT ON;

SELECT orderid, orderdate, shippeddate, custid, empid, shipperid
FROM Sales.Orders
WHERE (orderid   = @orderid   OR @orderid   IS NULL)
  AND (orderdate = @orderdate OR @orderdate IS NULL)
  AND (custid    = @custid    OR @custid    IS NULL)
  AND (empid     = @empid     OR @empid     IS NULL);
GO
```

Observe the header of the stored procedure with the definition of the input parameters and the syntax for defining default values. Some people like to place the parameter definitions within parentheses, as in:

```
CREATE OR ALTER PROC dbo.GetOrders
(
  @orderid   AS INT  = NULL,
  @orderdate AS DATE = NULL,
  @custid    AS INT  = NULL,
  @empid     AS INT  = NULL
)
AS...
```

As for the body of the stored procedure, notice that there's no mandatory BEGIN-END block like in multistatement user-defined functions, but you can use one if that's your styling preference, as in:

```
BEGIN

  SET XACT_ABORT, NOCOUNT ON;

  SELECT orderid, orderdate, shippeddate, custid, empid, shipperid
  FROM Sales.Orders
  WHERE (orderid   = @orderid   OR @orderid   IS NULL)
    AND (orderdate = @orderdate OR @orderdate IS NULL)
    AND (custid    = @custid    OR @custid    IS NULL)
    AND (empid     = @empid     OR @empid     IS NULL);

END;
```

The stored procedure's code starts by setting the options XACT_ABORT and NOCOUNT to ON. The XACT_ABORT option determines the effect of run-time errors raised by T-SQL statements. When this option is OFF (the default in most cases), some errors cause an open transaction to roll back and the execution of the code to be aborted, whereas other errors leave the transaction open. To get a more reliable and consistent behavior, I consider it a best practice to set this option to ON, and this way all errors cause an open transaction to be rolled back and the execution of the code to be aborted. The NOCOUNT option suppresses messages indicating how many rows were affected by data manipulation statements. When it's OFF (the default), those messages can degrade query performance due to the network traffic that they generate, plus this causes trouble for client applications that perceive those as query results.

The code then invokes a query against the Sales.Orders table that filters the data based on the specified parameters. For each parameter the query's WHERE clause has the following *disjunction* of predicates (predicates separated by an OR operator):

```
column = @parameter OR @parameter IS NULL
```

When the user doesn't specify an input value, the parameter is set to the default NULL. In such a case the predicate @parameter IS NULL is true, and therefore the disjunction of predicates is true, so no filtering is applied. When the user does specify an input value, the predicate @parameter IS NULL is false, and it is left to the predicate column = @parameter to determine whether to keep the row or discard it.

Run the following code to test the procedure, asking to filter orders placed on November 11, 2015 by customer 85:

```
EXEC dbo.GetOrders @orderdate = '20151111', @custid = 85;
```

This code generates the following output:

```
orderid     orderdate  shippeddate custid      empid       shipperid
----------- ---------- ----------- ----------- ----------- -----------
10737       2015-11-11 2015-11-18  85          2           2
```

Note that you can also execute the procedure, passing input values without specifying the target parameter names, but then you need to know the positions of the parameters in the procedure's header. Here's the equivalent to the above procedure execution passing inputs by position:

```
EXEC dbo.GetOrders DEFAULT, '20151111', 85, DEFAULT;
```

Using named parameters is considered a best practice because you need to specify only the applicable parameters, and you don't need to know in what order they are defined in the procedure's header. The code is also much clearer.

Execute the procedure again, asking to filter the order with order ID 42:

```
EXEC dbo.GetOrders @orderid = 42;
```

Such an order doesn't exist and therefore the procedure returns an empty set:

```
orderid     orderdate  shippeddate custid      empid       shipperid
----------- ---------- ----------- ----------- ----------- -----------
```

Stored procedures and dynamic SQL

The previous section demonstrated using a stored procedure to handle a dynamic search conditions task to query and filter data from the Sales.Orders table using a static query. This section describes an alternative solution that uses *dynamic SQL*, which is a technique that involves building a batch of code as a character string, usually in a variable, and then telling SQL Server to execute the code that resides in that variable.

Using a static query in our case is not ideal in terms of query performance. The reason for this has to do with the fact that SQL Server caches the execution plan for the query for reuse in subsequent executions of the stored procedure. The cached plan has to incorporate all filter predicates and not just the ones that are related to the parameters that are applicable in the execution that gets optimized. Otherwise, SQL Server would not be able to reuse the plan in subsequent executions that specify a different set of applicable parameters. This tends

to result in a generalized plan that cannot really be optimal for all possible combinations of parameters.

One way to get efficient plans is to add the query option RECOMPILE, as in:

```
<query> OPTION(RECOMPILE);
```

With this option, in every execution of the stored procedure SQL Server optimizes the query from scratch, after applying *parameter embedding* (replacing the parameters with constants) and normalizing the query to remove the redundant parts. For instance, if you execute the procedure, providing an input value only to the @orderid parameter (say, 10248), effectively the query that gets optimized is:

```
SELECT orderid, orderdate, shippeddate, custid, empid, shipperid
FROM Sales.Orders
WHERE orderid   = 10248;
```

As you can realize, the likelihood to get efficient plans with this solution is quite high. But this comes at the cost of recompiling the query in every execution of the stored procedure. If you want to both get optimal plans and optimal cached query plan reuse behavior, you need to use a different solution. One of the ways to achieve this is to dynamically build the query string with only the relevant parameterized predicates, and execute the code that you built with the sp_executesql procedure. Here's the solution code that implements this approach:

```
CREATE OR ALTER PROC dbo.GetOrders
  @orderid   AS INT  = NULL,
  @orderdate AS DATE = NULL,
  @custid    AS INT  = NULL,
  @empid     AS INT  = NULL
AS

SET XACT_ABORT, NOCOUNT ON;

DECLARE @sql AS NVARCHAR(MAX) = N'SELECT orderid, orderdate, shippeddate, custid, empid,
shipperid
FROM Sales.Orders
WHERE 1 = 1'
  + CASE WHEN @orderid   IS NOT NULL THEN N' AND orderid   = @orderid  ' ELSE N'' END
  + CASE WHEN @orderdate IS NOT NULL THEN N' AND orderdate = @orderdate' ELSE N'' END
  + CASE WHEN @custid    IS NOT NULL THEN N' AND custid    = @custid   ' ELSE N'' END
  + CASE WHEN @empid     IS NOT NULL THEN N' AND empid     = @empid    ' ELSE N'' END
  + N';'

EXEC sys.sp_executesql
  @stmt = @sql,
  @params = N'@orderid AS INT, @orderdate AS DATE, @custid AS INT, @empid AS INT',
  @orderid   = @orderid,
  @orderdate = @orderdate,
  @custid    = @custid,
  @empid     = @empid;
GO
```

The code declares a variable called @sql to store the dynamic query string. It constructs it with the fixed SELECT and FROM parts first. It starts the WHERE clause with the predicate 1 = 1, which is always true, to avoid needing to deal differently with the first applicable element that needs to be concatenated versus the nonfirst ones. Then, using a CASE expression, for each applicable parameter (one that is not NULL), the code concatenates a parameterized predicate to the filter, otherwise an empty string.

The code then executes the dynamically built query by using the sp_executesql procedure. The first part of the procedure is the @stmt parameter, where you provide the string with the code that you want to execute (@sql variable in our case). The second part is the @params parameter, where you declare all of the dynamic batch parameters. In our case, we declare a corresponding dynamic batch parameter for each of the stored procedure's parameters. It's quite all right if you wish to use the same names for the dynamic batch parameters and the stored procedure's parameters as is done in our example. The last part is a series of assignments of the stored procedure's parameters to the corresponding dynamic batch parameters.

Use the following code to test the stored procedure:

```
EXEC dbo.GetOrders @orderdate = '20151111', @custid = 85;
```

You get the following output:

```
orderid     orderdate  shippeddate custid      empid       shipperid
----------- ---------- ----------- ----------- ----------- -----------
10737       2015-11-11 2015-11-18  85          2           2
```

Each unique query string (same combination of parameters) that is used gets optimized separately, and SQL Server is able to reuse a previously cached plan when the query string matches.

There's a certain caveat related to security when using dynamic SQL. Earlier I mentioned that SQL Server supports a security model where you grant users EXECUTE permissions on a stored procedure without granting them direct permissions against the underlying objects. This way users are able to perform the task only through the stored procedure and not directly. This capability is known as *ownership chaining*. However, ownership chaining is limited only to SELECT, INSERT, UPDATE, and DELETE statements using static SQL, and only when the owner of the calling and the called objects is the same. Because in our case the code is executed using dynamic SQL, ownership chaining doesn't apply and the executing user needs direct SELECT permissions against the Sales.Orders table. Before I provide a workaround, I demonstrate the fact that when using dynamic SQL, only EXECUTE permission on the stored procedure isn't sufficient.

Run the following code to create a login called login1, and an associated user in the database called user1:

```
CREATE LOGIN login1 WITH PASSWORD = 'J345#$)thb';
GO
CREATE USER user1 FOR LOGIN login1;
GO
```

Next, run the following code to grant EXECUTE permission on the stored procedure to user1:

```
GRANT EXEC ON dbo.GetOrders TO user1;
```

Use the following code to display the current execution context:

```
SELECT SUSER_NAME() AS [login], USER_NAME() AS [user];
```

You should get output similar to the following, with your login name, of course:

```
login                          user
------------------------------ -----
MicrosoftAccount\<your acount>  dbo
```

Run the following code to set the execution context to login1:

```
EXECUTE AS LOGIN = 'login1';
```

Run the following code to display the current execution context again:

```
SELECT SUSER_NAME() AS [login], USER_NAME() AS [user];
```

You get the following output indicating that your context indeed changed to login1 as the login, and user1 as the database user:

```
login   user
------- ------
login1  user1
```

Run the following code in attempt to execute the stored procedure:

```
EXEC dbo.GetOrders @orderdate = '20151111', @custid = 85;
```

You get the following permission error:

```
Msg 229, Level 14, State 5, Line 882
The SELECT permission was denied on the object 'Orders', database 'TSQLV4', schema
'Sales'.
```

Run the following code to revert back to original execution context:

```
REVERT;
```

As a workaround, you can define the stored procedure with an EXECUTE AS clause, to impersonate the security context of the procedure's execution to that of the specified entity. For example, using the option EXECUTE AS OWNER impersonates the security context to that of the stored procedure's owner. Alternatively, you can provide a specific user name that has the right permissions. Run the following code to recreate the stored procedure using the EXECUTE AS OWNER option:

```
CREATE OR ALTER PROC dbo.GetOrders
  @orderid   AS INT  = NULL,
  @orderdate AS DATE = NULL,
  @custid    AS INT  = NULL,
  @empid     AS INT  = NULL
```

```
WITH EXECUTE AS OWNER
AS

SET XACT_ABORT, NOCOUNT ON;

DECLARE @sql AS NVARCHAR(MAX) = N'SELECT orderid, orderdate, shippeddate, custid, empid,
shipperid
FROM Sales.Orders
WHERE 1 = 1'
  + CASE WHEN @orderid   IS NOT NULL THEN N' AND orderid   = @orderid  ' ELSE N'' END
  + CASE WHEN @orderdate IS NOT NULL THEN N' AND orderdate = @orderdate' ELSE N'' END
  + CASE WHEN @custid    IS NOT NULL THEN N' AND custid    = @custid   ' ELSE N'' END
  + CASE WHEN @empid     IS NOT NULL THEN N' AND empid     = @empid    ' ELSE N'' END
  + N';'

EXEC sys.sp_executesql
  @stmt = @sql,
  @params = N'@orderid AS INT, @orderdate AS DATE, @custid AS INT, @empid AS INT',
  @orderid   = @orderid,
  @orderdate = @orderdate,
  @custid    = @custid,
  @empid     = @empid;
GO
```

Run the following code to set the execution context to login1:

```
EXECUTE AS LOGIN = 'login1';
```

Try to execute the stored procedure again:

```
EXEC dbo.GetOrders @orderdate = '20151111', @custid = 85;
```

This time the stored procedure runs successfully.

Run the following code to revert back to original execution context:

```
REVERT;
```

MORE INFO **DYNAMIC SQL AND SECURITY IN STORED PROCEDURES**

The aforementioned method to impersonate the security context of the procedure's execution is quite simple and straightforward, but not always ideal in terms of auditing and monitoring capabilities. There are more complex, yet more recommended alternatives, in which you sign the stored procedure with a certificate. You associate the certificate with a user that cannot login, and grant that user with the appropriate permissions. For more information on the topic see Erland Sommarskog's text "The Curse and Blessings of Dynamic SQL" at *http://www.sommarskog.se/dynamic_sql.html* and "Giving Permissions through Stored Procedures" at *http://www.sommarskog.se/grantperm.html*.

Using output parameters and modifying data

If you need to return scalar values back from a stored procedure, you can do this by using output parameters. Also, unlike with user-defined functions, you can modify data in the database from stored procedures. This section demonstrates both capabilities.

To define a parameter as an output one add the keyword OUTPUT or OUT in its definition. Also, when executing the procedure, you need to provide a local variable to accept the returned value, and indicate the keyword OUTPUT or OUT again in that assignment, otherwise the parameter is actually treated as an input one.

As an example, suppose that you need to develop a solution for generating integer keys with a guarantee for no gaps between the values. For instance, perhaps you need such a solution to generate invoice numbers and you need an assurance that they are to be consecutive. You cannot use the identity property or the sequence object because these solutions do not guarantee that you won't have gaps between the keys. If you create an identity or sequence value in a transaction that ends up failing, that value is gone.

You decide to handle the task by creating a table called MySequences where you maintain your own custom sequences. For each such sequence you store a row with the sequence name and the last used value. Whenever you need a new key, you update the relevant row to increase the current value, query it, and use it in the target table. When you update a row, your session obtains an exclusive lock on the row and keeps the lock until the end of the transaction. This means that on one hand no one else can obtain a new value until your transaction is over; on the other hand, if the transaction fails, the current sequence value is undone to the original one. This way you have a guarantee for no gaps.

Use the following code to create the table dbo.MySequences:

```
DROP TABLE IF EXISTS dbo.MySequences;
GO
CREATE TABLE dbo.MySequences
(
  seqname VARCHAR(128) NOT NULL
    CONSTRAINT PK_MySequences PRIMARY KEY,
  val INT NOT NULL
    CONSTRAINT DFT_MySequences_val DEFAULT(0)
);
```

Run the following code to add a row representing a sequence for generating invoice numbers:

```
INSERT INTO dbo.MySequences(seqname, val) VALUES('SEQINVOICES', 0);
```

You name the custom sequence SEQINVOICES and initialize it with 0 so that the first value that it generates is 1.

Use the following code to create a stored procedure called GetSequenceValue to handle a request for a new value for a given sequence name:

```
CREATE OR ALTER PROC dbo.GetSequenceValue
  @seqname AS VARCHAR(128),
  @val     AS INT OUTPUT
AS

SET XACT_ABORT, NOCOUNT ON;

UPDATE dbo.MySequences
  SET @val = val += 1
WHERE seqname = @seqname;

IF @@ROWCOUNT = 0
  THROW 51001, 'Specified sequence was not found.', 1;
GO
```

Observe that the header of the stored procedure defines an input parameter called @seqname to accept the input sequence name, and an output parameter called @val to return the newly generated sequence value. The procedure uses an UPDATE statement with a variable (covered in Chapter 1) to increment the requested sequence value by 1, and assigns the new value to the output parameter. If the requested custom sequence doesn't exist, the UPDATE statement has affected zero rows, in which case the stored procedure throws an error using the THROW command.

Use the following code to request a new invoice number:

```
DECLARE @newinvoicenumber AS INT;
EXEC dbo.GetSequenceValue @seqname = 'SEQINVOICES', @val = @newinvoicenumber OUTPUT;
SELECT @newinvoicenumber AS newinvoicenumber;
```

The code uses a local variable to absorb the returned value from the output parameter. Notice that you specify the OUTPUT keyword again in the execution; otherwise the parameter is treated as an input one.

This code generates the following output:

```
newinvoicenumber
----------------
1
```

Run the code a couple of more times, and see how the sequence value keeps advancing.

Try running the code with a sequence name that doesn't exist:

```
DECLARE @newinvoicenumber AS INT;
EXEC dbo.GetSequenceValue @seqname = 'NOSUCHSEQUENCE', @val = @newinvoicenumber OUTPUT;
SELECT @newinvoicenumber AS newinvoicenumber;
```

This time the procedure throws an error:

```
Msg 51001, Level 16, State 1, Procedure dbo.GetSequenceValue, Line 15 [Batch Start Line
978]
Specified sequence was not found.
```

Later in the chapter, Skill 3.2 covers error handling.

Using cursors

Another example where the encapsulation provided by stored procedures is beneficial is when using cursors. A cursor allows you to iterate through rows of some query result one at a time. Solutions that use cursors tend to be lengthy because you need explicit code to define the cursor, open it, iterate through its rows, and apply some logic per row, close, and deallocate it. With a stored procedure, you can hide all that complexity. Furthermore, if at a later point you manage to find a better solution, perhaps one that doesn't use a cursor, you can simply alter the stored procedure's implementation. This is transparent to the users of the stored procedure.

The following example demonstrates the use of a stored procedure with a cursor. It involves a task called *depleting quantities*. You're given a table called Transactions, which you create and populate by running the following code:

```
DROP TABLE IF EXISTS dbo.Transactions;
GO
CREATE TABLE dbo.Transactions
(
  txid INT NOT NULL CONSTRAINT PK_Transactions PRIMARY KEY,
  qty  INT NOT NULL,
  depletionqty INT NULL
);
GO

TRUNCATE TABLE dbo.Transactions;
INSERT INTO dbo.Transactions(txid, qty)
  VALUES(1,2),(2,5),(3,4),(4,1),(5,10),(6,3),(7,1),(8,2),(9,1),(10,2),(11,1),(12,9);
```

Your task is to create a stored procedure called ComputeDepletionQuantities that processes the transactions that are stored in the Transactions table. The procedure should process the transactions in order based on the txid column. Each transaction adds a certain quantity (qty column) of some item to a container. The container has a limited capacity, which is provided as an input parameter called @maxallowedqty to the stored procedure. As soon as the cumulative quantity in the container exceeds the maximum allowed, the container needs to be depleted, and your code should write the depletion quantity to the depeletionqty column of the current transaction. At the end of the stored procedure's execution, rows in the Transactions table representing transactions where the container was depleted should have the depletion quantity stored in the depleteqty column, and all remaining rows should have a NULL in that column. Furthermore, the stored procedure should return a result set showing the transactions in order, along with the depletion quantity (depleteqty column) where applicable, and the current cumulative quantity in the container (totalqty column).

After you're done developing the stored procedure, use the following code to test it:

```
EXEC dbo.ComputeDepletionQuantities @maxallowedqty = 5;
```

The stored procedure should return the following output for the given sample data in the Transactions table and the specified maximum container capacity of 5.

```
txid        qty         depletionqty totalqty
----------- ----------- ------------ -----------
1           2           NULL         2
2           5           7            0
3           4           NULL         4
4           1           NULL         5
5           10          15           0
6           3           NULL         3
7           1           NULL         4
8           2           6            0
9           1           NULL         1
10          2           NULL         3
11          1           NULL         4
12          9           13           0
```

The following code demonstrates one way to implement the stored procedure using a cursor:

```
CREATE OR ALTER PROC dbo.ComputeDepletionQuantities
  @maxallowedqty AS INT
AS

SET XACT_ABORT, NOCOUNT ON;

UPDATE dbo.Transactions
  SET depletionqty = NULL
WHERE depletionqty IS NOT NULL;

DECLARE @qty AS INT, @sumqty AS INT = 0;

DECLARE C CURSOR FOR
  SELECT qty
  FROM dbo.Transactions
  ORDER BY txid;

OPEN C;

FETCH NEXT FROM C INTO @qty;

WHILE @@FETCH_STATUS = 0
BEGIN
  SET @sumqty += @qty;

  IF @sumqty > @maxallowedqty
  BEGIN
    UPDATE dbo.Transactions
      SET depletionqty = @sumqty
    WHERE CURRENT OF C;

    SET @sumqty = 0;
  END;

  FETCH NEXT FROM C INTO @qty;
END;
```

```
CLOSE C;

DEALLOCATE C;

SELECT txid, qty, depletionqty,
  SUM(qty - ISNULL(depletionqty, 0))
    OVER(ORDER BY txid ROWS UNBOUNDED PRECEDING) AS totalqty
FROM dbo.Transactions
ORDER BY txid;
GO
```

The code starts by updating the Transactions table, setting the depletionqty column to NULL where it currently isn't NULL.

Next, the code declares local variables called @qty and @sumqty to hold the current transaction's quantity and current cumulative quantity (initialized with 0), respectively.

The code then declares a cursor based on a query that returns the transaction quantities in order, opens it, and fetches the first transaction's quantity into the @qty variable.

Next, the code executes a loop that iterates per transaction while the last fetch was successful (@@FETCH_STATUS function returns 0, meaning we haven't reached the end of the cursor yet). In each iteration of the loop, the code increases the current cumulative quantity (@sumqty variable) by the current transaction's quantity (@qty variable). If the new cumulative quantity exceeds the maximum allowed in the container, the code applies two steps. One, it uses the syntax UPDATE <table> WHERE CURRENT OF <cursor> to update the depletion quantity of the Transactions table's row that the cursor is currently positioned on to the new cumulative quantity. Two, it resets the current cumulative quantity to 0. Before finishing the current iteration of the loop, the code fetches the next transaction's quantity into the @qty variable.

After the loop is done with all of its iterations, the code closes and deallocates the cursor.

At this point the Transactions table is updated with the correct depletion quantities in the applicable transactions. The code finally issues a query that in addition to returning the current data from the Transactions table, also computes the current total quantity in the container (totalqty column) after each transaction. The query uses the following expression based on a window function to compute the totalqty result column:

```
SUM(qty - ISNULL(depletionqty, 0))
  OVER(ORDER BY txid ROWS UNBOUNDED PRECEDING) AS totalqty
```

The computation is a running total of all transaction quantities minus the depletion quantities (or zero if NULL) based on txid ordering, from the beginning of the activity and up to the current transaction.

Run the following code to test the stored procedure with a maximum allowed capacity of 5:

```
EXEC dbo.ComputeDepletionQuantities @maxallowedqty = 5;
```

Verify that you get the correct result, which was provided earlier. Of course, feel free to test the procedure with different input values.

FURTHER READING **TRIGGERS**

A trigger, in a way, is a special kind of stored procedure. You associate a trigger with an event, such as an insert against a table, and the trigger's code runs automatically whenever such an event takes place. You can use triggers for purposes such as auditing, enforcing complex integrity rules that cannot be enforced with constraints, and more. Triggers are outside the scope of this book because they are not part of the exam's skill set. If you're interested in information about triggers, see the online documentation at *https://msdn.microsoft.com/en-us/library/ms189799.aspx*.

When you're done, run the following code for cleanup:

```
DROP USER IF EXISTS user1;
GO
DROP LOGIN login1;
GO
DROP PROC IF EXISTS dbo.GetOrders, dbo.GetSequenceValue, dbo.ComputeDepletionQuantities;
DROP TABLE IF EXISTS dbo.MySequences, dbo.Transactions;
```

Skill 3.2: Implement error handling and transactions

Transactions and error handling are two mechanisms in SQL Server that enable you to work with a consistent database and define the course of action to take in case of errors. The two are strongly intertwined. When errors happen in transactions, the default handling of SQL Server is not always the desired one; with your own error handling, you have some degree of control over the outcome. This skill starts with coverage of transactions. It then describes the error handling constructs that T-SQL supports. It then explains how to handle errors that happen in transactions.

This section covers how to:

- Determine results of Data Definition Language (DDL) statements based on transaction control statements
- Implement TRY...CATCH error handling with Transact-SQL
- Generate error messages with THROW and RAISERROR
- Implement transaction control in conjunction with error handling in stored procedures

FURTHER READING **ERROR HANDLING**

Much of what I learned about error handling I learned from fellow Data Platform MVP Erland Sommarskog. Erland spent a lifetime creating and maintaining detailed texts covering various important topics related to SQL Server. His texts are considered the go to resource for the subject matter, and are worth their weight in gold. I cannot give you better advice in terms of learning about error handling in SQL Server than suggesting that besides going over the material in this book, you read his articles on the topic. You can find Erland's texts in his website at *http://www.sommarskog.se*. He wrote a multi-part series on error handling. Make sure that at minimum you read Part One – Jumpstart Error Handling at *http://www.sommarskog.se/error_handling/Part1.html* and Part Two – Commands and Mechanisms at *http://www.sommarskog.se/error_handling/Part2.html*.

Understanding transactions

A transaction is a unit of work with one or more activities that manipulate data, and possibly its structure (yes, unlike in some other database platforms, in SQL Server most DDL is transactional!). A transaction has, or at least should have, four main properties known collectively as the *ACID* properties. A stands for atomicity, C for consistency, I for isolation, and D for durability.

Theoretically, a transaction should be *atomic*; namely either complete in its entirety or not take place at all. In practice, this is not always the default behavior in SQL Server, and in order to achieve true atomicity, you need to add your own error handling code.

A transaction should be *consistent*; namely, it should transition the database from one consistent state to another in terms of adhering to the data model, constraints, and triggers.

A transaction should be *isolated*; this means that intermediate inconsistent states of the data are supposed to be visible only to the transaction that made the changes, but not to other transactions. You can set what's called an *isolation level* either at the session level with a SET TRANSACTION ISOLATION LEVEL option, or at the query level with a table hint to control the degree of isolation that you get.

MORE INFO ON ISOLATION LEVELS

You can find more information on isolation levels at *https://technet.microsoft.com/en-us/library/ms189122(v=sql.105).aspx*.

Finally, a transaction should be durable; this means that when you commit the transaction and get an acknowledge from the database that the transaction committed successfully, you can rest assured that the transaction's changes are durable. This means that the changes can survive a crash of the SQL Server process, such as a result of a power failure event.

This section explains how to define transactions, nesting of transactions, and working with savepoints.

Defining transactions

SQL Server allows you to either explicitly define the transaction's boundaries yourself or to let it define those implicitly for you. To explicitly mark the beginning of a transaction, use the BEGIN TRANSACTION statement (or BEGIN TRAN for brevity). To end the transaction and commit its work, use the COMMIT TRANSACTION statement (supported alternatives: COMMIT TRAN, COMMIT WORK and just COMMIT). To end a transaction and roll back its work, undoing all of its changes, use the ROLLBACK TRANSACTION statement (supported alternatives: ROLLBACK TRAN, ROLLBACK WORK and just ROLLBACK).

You can query a function called @@TRANCOUNT to know whether you're currently in an open transaction or not. If you're in an open transaction the function returns a value greater than zero, otherwise, it returns zero. I provide more details about this function later under the topic Nesting of transactions.

As an example, the following code uses an explicit user transaction to add a new order to the TSQLV4 sample database:

```
USE TSQLV4;
SET XACT_ABORT, NOCOUNT ON;

-- start a new transaction
BEGIN TRAN;

-- declare a variable
DECLARE @neworderid AS INT;

-- insert a new order into the Sales.Orders table
INSERT INTO Sales.Orders
    (custid, empid, orderdate, requireddate, shippeddate,
     shipperid, freight, shipname, shipaddress, shipcity,
     shippostalcode, shipcountry)
  VALUES
    (1, 1, '20170212', '20170301', '20170216',
     1, 10.00, N'Shipper 1', N'Address AAA', N'City AAA',
     N'11111', N'Country AAA');

-- save the new order id in the variable @neworderid
SET @neworderid = SCOPE_IDENTITY();

PRINT 'Added new order header with order ID ' + CAST(@neworderid AS VARCHAR(10))
  + '. @@TRANCOUNT is ' + CAST(@@TRANCOUNT AS VARCHAR(10)) + '.';

-- insert order lines for new order into Sales.OrderDetails
INSERT INTO Sales.OrderDetails(orderid, productid, unitprice, qty, discount)
  VALUES(@neworderid, 1, 10.00, 1, 0.000),
        (@neworderid, 2, 10.00, 1, 0.000),
        (@neworderid, 3, 10.00, 1, 0.000);

PRINT 'Added order lines to new order. @@TRANCOUNT is '
  + CAST(@@TRANCOUNT AS VARCHAR(10)) + '.';

-- commit the transaction
COMMIT TRAN;
```

The examples in this section use ad-hoc batches for simplicity, but in a later section you will encapsulate the code in a stored procedure. The code starts by setting the XACT_ABORT and NOCOUNT options to ON. As a reminder, when the XACT_ABORT setting is off (the default), not all run-time errors cause the transaction to roll back, and execution of the code to abort. By setting this option to on, you provide a more consistent and expected behavior from transactions whereby all errors cause the transaction to roll back and execution of the code to abort. By setting the NOCOUNT option to ON you request to suppress messages reporting how many rows were affected by DML statements.

The code uses a single transaction to add both an order header to the Sales.Orders table, and corresponding order lines to the Sales.OrderDetails table. After inserting the order header row to Orders, the code saves the order ID that was just generated by the identity property to a variable, and then uses the variable when adding the order line rows to OrderDetails.

This code prints the state of the transaction after each of the INSERT statements, generating the following output:

```
Added new order header with order ID 11078. @@TRANCOUNT is 1.
Added order lines to new order. @@TRANCOUNT is 1.
```

In case you already ran code that added orders previously, the new order ID that SQL Server creates for your new order may be different than in the earlier example. Remember that you can always run the code that creates the sample database TSQLV4 to start with a clean copy.

Remember that because you set the XACT_ABORT setting to ON, if there had been a run-time error after any of the INSERT statements, the transaction would have rolled back in its entirety and execution of the code would have been aborted. In our case, the transaction completed successfully. If you now query the Orders and OrderDetails tables, you find the data for the new order, including both the order header and its corresponding order lines. Run the following code to achieve this (use the order ID that you got in case it's different than the one in this example):

```
SELECT orderid, orderdate, custid, empid
FROM Sales.Orders
WHERE orderid = 11078;

SELECT orderid, productid, qty
FROM Sales.OrderDetails
WHERE orderid = 11078;
```

This code generates the following output:

```
orderid     orderdate  custid      empid
----------- ---------- ----------- -----------
11078       2017-02-12 1           1

orderid     productid   qty
----------- ----------- ------
11078       1           1
11078       2           1
11078       3           1
```

Next, run the following code to try adding an order with an invalid order line where the discount is greater that 1:

```
SET XACT_ABORT, NOCOUNT ON;

BEGIN TRAN;

DECLARE @neworderid AS INT;

INSERT INTO Sales.Orders
    (custid, empid, orderdate, requireddate, shippeddate,
     shipperid, freight, shipname, shipaddress, shipcity,
     shippostalcode, shipcountry)
  VALUES
    (2, 2, '20170212', '20170301', '20170216',
     2, 20.00, N'Shipper 2', N'Address BBB', N'City BBB',
     N'22222', N'Country BBB');

SET @neworderid = SCOPE_IDENTITY();

PRINT 'Added new order header with order ID ' + CAST(@neworderid AS VARCHAR(10))
  + '. @@TRANCOUNT is ' + CAST(@@TRANCOUNT AS VARCHAR(10)) + '.';

INSERT INTO Sales.OrderDetails(orderid, productid, unitprice, qty, discount)
  VALUES(@neworderid, 1, 20.00, 2, 2.000), -- CHECK violation since discount > 1
        (@neworderid, 2, 20.00, 2, 0.000),
        (@neworderid, 3, 20.00, 2, 0.000);

PRINT 'Added order lines to new order. @@TRANCOUNT is '
  + CAST(@@TRANCOUNT AS VARCHAR(10)) + '.';

COMMIT TRAN;
```

The first INSERT statement, which adds the order header row, completes successfully. However, the second INSERT statement, which attempts to add the order lines, fails due to a CHECK constraint violation because one of the order lines has a discount value that is greater than 1. The table has a CHECK constraint that limits the discount to the range 0 through 1. You can see the constraints that are defined on the table by running the following code:

```
EXEC sys.sp_helpconstraint 'Sales.OrderDetails';
```

Because the XACT_ABORT setting is turned on, the error causes the transaction to roll back and execution of the code to abort. The execution of the code generates the following output:

```
Added new order header with order ID 11079. @@TRANCOUNT is 1.
Msg 547, Level 16, State 0, Line 86
The INSERT statement conflicted with the CHECK constraint "CHK_discount". The conflict
occurred in database "TSQLV4", table "Sales.OrderDetails", column 'discount'.
```

Query both tables looking for the rows that are related to the new order:

```
SELECT orderid, orderdate, custid, empid
FROM Sales.Orders
```

```
WHERE orderid = 11079;

SELECT orderid, productid, qty
FROM Sales.OrderDetails
WHERE orderid = 11079;
```

You get empty sets back from both queries:

```
orderid     orderdate  custid      empid
----------- ---------- ----------- -----------

orderid     productid   qty
----------- ----------- ------
```

If you don't explicitly define the transaction boundaries, SQL Server uses an *autocommit* mode where each individual statement is considered a separate transaction, as if it's preceded by a BEGIN TRAN statement and followed by a COMMIT TRAN statement. Note that with most statements, the individual statement must be atomic, meaning that the single statement either completes in its entirety or not take place at all. For instance, suppose that you issue a DELETE statement that deletes 100 rows from a table under autocommit mode. If the statement fails before completion, say after 17 rows were deleted, SQL Server undoes the change. Either all 100 rows get deleted, or none at all.

EXAM TIP

The types of statements that are considered *transactional statements* include DML statements (such as SELECT against a table, INSERT, UPDATE, DELETE, TRUNCATE, MERGE), many DDL statements such as creating, altering and dropping tables, DCL statements like GRANT and REVOKE, and others. Assigning values to variables as well as modifying data in table variables are not transactional operations, so if you applied such activities in a transaction that ended up rolling back, those activities are not undone.

In our example of adding an order, it's not a good idea to use the autocommit mode because under this mode the statements that add the order header and order lines are treated as two separate transactions. If the first succeeds and the second fails, you end up with the order header added without related order lines. The following code demonstrates this:

```
SET XACT_ABORT, NOCOUNT ON;

DECLARE @neworderid AS INT;

INSERT INTO Sales.Orders
    (custid, empid, orderdate, requireddate, shippeddate,
     shipperid, freight, shipname, shipaddress, shipcity,
     shippostalcode, shipcountry)
  VALUES
    (3, 3, '20170212', '20170301', '20170216',
     3, 30.00, N'Shipper 3', N'Address CCC', N'City CCC',
     N'33333', N'Country CCC');

SET @neworderid = SCOPE_IDENTITY();
```

```
PRINT 'Added new order header with order ID ' + CAST(@neworderid AS VARCHAR(10))
  + '. @@TRANCOUNT is ' + CAST(@@TRANCOUNT AS VARCHAR(10)) + '.';

INSERT INTO Sales.OrderDetails(orderid, productid, unitprice, qty, discount)
  VALUES(@neworderid, 1, 30.00, 3, 2.000), -- CHECK violation since discount > 1
        (@neworderid, 2, 30.00, 3, 0.000),
        (@neworderid, 3, 30.00, 3, 0.000);

PRINT 'Added order lines to new order. @@TRANCOUNT is '
  + CAST(@@TRANCOUNT AS VARCHAR(10)) + '.';
```

This code generates the following output:

```
Added new order header with order ID 11080. @@TRANCOUNT is 0.
Msg 547, Level 16, State 0, Line 126
The INSERT statement conflicted with the CHECK constraint "CHK_discount". The conflict
occurred in database "TSQLV4", table "Sales.OrderDetails", column 'discount'.
```

Notice that this time you were not in an open transaction after adding the order header row. Query the tables to see if any rows related to the new order survived:

```
SELECT orderid, orderdate, custid, empid
FROM Sales.Orders
WHERE orderid = 11080;

SELECT orderid, productid, qty
FROM Sales.OrderDetails
WHERE orderid = 11080;
```

This code generates the following output:

```
orderid     orderdate  custid      empid
----------- ---------- ----------- -----------
11080       2017-02-12 3           3

orderid     productid   qty
----------- ----------- ------
```

Observe that the order header row exists, but there are no corresponding order lines. That's obviously an undesirable outcome, which is why it's important to perform both activities in a single transaction and not in two separate ones.

The SQL standard defines an *implicit transactions* mode, which is supposed to be the default mode for working with transactions. Under this mode, when you issue a transactional statement, if a transaction is not open at that point, the system is supposed to open an implicit transaction for you. However, unlike under the default autocommit mode in SQL Server, under the standard implicit transactions mode you are responsible for explicitly closing the transaction by either committing it or rolling it back. You enable the standard implicit transactions mode by setting the session option IMPLICIT_TRANSACTIONS to ON, like so (make sure you run this code for the next demo to work):

```
SET IMPLICIT_TRANSACTIONS ON;
```

Assuming you enabled this option, run the following code to add another order to the system:

```
SET XACT_ABORT, NOCOUNT ON;

DECLARE @neworderid AS INT;

-- following statement triggers the opening of a transaction but doesn't close it
INSERT INTO Sales.Orders
    (custid, empid, orderdate, requireddate, shippeddate,
     shipperid, freight, shipname, shipaddress, shipcity,
     shippostalcode, shipcountry)
  VALUES
    (4, 4, '20170212', '20170301', '20170216',
     1, 40.00, N'Shipper 1', N'Address AAA', N'City AAA',
     N'11111', N'Country AAA');

SET @neworderid = SCOPE_IDENTITY();

PRINT 'Added new order header with order ID ' + CAST(@neworderid AS VARCHAR(10))
  + '. @@TRANCOUNT is ' + CAST(@@TRANCOUNT AS VARCHAR(10)) + '.';

INSERT INTO Sales.OrderDetails(orderid, productid, unitprice, qty, discount)
  VALUES(@neworderid, 1, 40.00, 4, 0.000),
        (@neworderid, 2, 40.00, 4, 0.000),
        (@neworderid, 3, 40.00, 4, 0.000);

PRINT 'Added order lines to new order. @@TRANCOUNT is '
  + CAST(@@TRANCOUNT AS VARCHAR(10)) + '.';

-- must explicitly commit the transaction
COMMIT TRAN;
```

Notice that this time there's no explicit BEGIN TRAN statement because you're relying on the implicit transactions mode, but there is an explicit COMMIT TRAN statement. This code generates the following output:

```
Added new order header with order ID 11081. @@TRANCOUNT is 1.
Added order lines to new order. @@TRANCOUNT is 1.
```

This time both the addition of the order header and the addition of the order lines ran in a single implicit transaction. If there had been a failure in any of the statements, the entire transaction would have rolled back. In our case, the transaction completed successfully. Query the tables to see the data that was added:

```
SELECT orderid, orderdate, custid, empid
FROM Sales.Orders
WHERE orderid = 11081;

SELECT orderid, productid, qty
FROM Sales.OrderDetails
WHERE orderid = 11081;
```

This code generates the following output:

```
orderid     orderdate  custid      empid
----------- ---------- ----------- -----------
11081       2017-02-12 4           4

orderid     productid   qty
----------- ----------- ------
11081       1           4
11081       2           4
11081       3           4
```

Run the following code to turn the implicit transactions mode back off:

```
SET IMPLICIT_TRANSACTIONS OFF;
```

Having implicit transactions turned on is usually a bad idea. It increases the likelihood that transactions stay open for long periods, or worse, just stay open until someone realizes that that's the case and manually intervenes. Long open transactions can cause performance problems in the system, including blocking and others. Also, your code has to be written differently to work with implicit transactions.

Remember that both DML and DDL are transactional in SQL Server. However, assignment of values to variables as well as changes to table variables are not transactional, other than each individual statement having to complete in its entirety or not take place at all. Run the following code to see that DDL is transactional:

```
DROP TABLE IF EXISTS dbo.T1;

BEGIN TRAN;

CREATE TABLE dbo.T1(col1 INT);
INSERT INTO dbo.T1(col1) VALUES(1),(2),(3);
PRINT 'In transaction';
SELECT col1 FROM dbo.T1;

ROLLBACK TRAN;

PRINT 'After transaction';
SELECT col1 FROM dbo.T1;
```

This code generates the following output under Results to Text (Ctrl + T):

```
In transaction
col1
-----------
1
2
3

After transaction
Msg 208, Level 16, State 1, Line 206
Invalid object name 'dbo.T1'.
```

As you can see, the table creation and population were both undone when the transaction was rolled back. As a result, the attempt to query the table after the ROLLBACK TRAN statement was submitted generates an error because the table doesn't exist at that point.

Run the following code to see that changes to a table variable are not undone when a transaction is rolled back:

```
BEGIN TRAN;

DECLARE @T1 AS TABLE(col1 INT);
INSERT INTO @T1(col1) VALUES(1),(2),(3);
PRINT 'In transaction';
SELECT col1 FROM @T1;

ROLLBACK TRAN;

PRINT 'After transaction';
SELECT col1 FROM @T1;
```

This code generates the following output, again, under Results to Text:

```
In transaction
col1
-----------
1
2
3

After transaction
col1
-----------
1
2
3
```

As you can see, the table variable itself, as well as the data that you inserted into it, are still present after the transaction was rolled back. As mentioned, you would get the same behavior when assigning values to regular variables in a transaction.

Nesting of transactions

SQL Server doesn't support a true concept of nested transactions where you can have an inner transaction that is completely independent of an outer one. Rather, in SQL Server you're either in a transaction or not. When you issue a BEGIN TRAN statement, SQL Server increases the value of @@TRANCOUNT in order to know when it needs to truly open a transaction—when @@TRANCOUNT changes from zero to a greater-than-zero value. Issuing a BEGIN TRAN statement when @@TRANCOUNT is already greater than zero has no real effect other than increasing the @@TRANCOUNT value. When you issue a COMMIT TRAN statement SQL Server decreases the @@TRANCOUNT value by 1. SQL Server truly commits the transaction only if you issue a COMMIT TRAN statement when @@TRANCOUNT is 1 prior to committing. However, if you issue a ROLLBACK TRAN statement in an open transaction, never mind the current value of @@TRANCOUNT, SQL Server rolls back the entire transaction.

Run the following code to test the behavior of nested BEGIN TRAN statement:

```
SET NOCOUNT ON;
DROP TABLE IF EXISTS dbo.T1;
GO
CREATE TABLE dbo.T1(col1 INT);

PRINT '@@TRANCOUNT before first BEGIN TRAN is '
  + CAST(@@TRANCOUNT AS VARCHAR(10)) + '.';

BEGIN TRAN;

PRINT '@@TRANCOUNT after first BEGIN TRAN is '
  + CAST(@@TRANCOUNT AS VARCHAR(10)) + '.';

BEGIN TRAN;

PRINT '@@TRANCOUNT after second BEGIN TRAN is '
  + CAST(@@TRANCOUNT AS VARCHAR(10)) + '.';

BEGIN TRAN;

PRINT '@@TRANCOUNT after third BEGIN TRAN is '
  + CAST(@@TRANCOUNT AS VARCHAR(10)) + '.';

INSERT INTO dbo.T1 VALUES(1),(2),(3);

COMMIT TRAN; -- this doesn't really commit

PRINT '@@TRANCOUNT after first COMMIT TRAN is '
  + CAST(@@TRANCOUNT AS VARCHAR(10)) + '.';

ROLLBACK TRAN; -- this does roll the transaction back

PRINT '@@TRANCOUNT after ROLLBACK TRAN is '
  + CAST(@@TRANCOUNT AS VARCHAR(10)) + '.';

SELECT col1 FROM dbo.T1;
```

This code generates the following output:

```
@@TRANCOUNT before first BEGIN TRAN is 0.
@@TRANCOUNT after first BEGIN TRAN is 1.
@@TRANCOUNT after second BEGIN TRAN is 2.
@@TRANCOUNT after third BEGIN TRAN is 3.
@@TRANCOUNT after first COMMIT TRAN is 2.
@@TRANCOUNT after ROLLBACK TRAN is 0.
col1
-----------
```

Observe that every BEGIN TRAN statement increased the @@TRANCOUNT value by 1, but as soon as the ROLLBACK TRAN statement was issued, @@TRANCOUNT dropped to 0. Also, observe that the INSERT statement that added three rows to the table T1 was rolled back even though you issued a COMMIT TRAN statement after it, because that statement did not really commit the transaction.

EXAM TIP

The point in supporting nested BEGIN TRAN-COMMIT TRAN statements is to allow you to issue those statements within a stored procedure without needing to check first if a transaction is already open. If your procedure actually opens the transaction, its COMMIT TRAN statement is supposed to also actually commit it as well. However, if a transaction was opened by an outer module or batch, it's not supposed to be the inner procedure's choice to actually commit it, rather the outer module's choice. Just make sure to remember that if you issue a ROLLBACK TRAN statement at any point, even if you do so from an inner module that was not the one that actually opened the transaction, this causes the entire transaction to truly roll back and all of its changes to be undone. The following example demonstrates this.

Normally, you would encapsulate the logic of the previously demonstrated task of adding an order within a stored procedure. You can pass the order header info to the stored procedure as scalar input parameters. However, because the order lines info should be passed as a set, you would probably want to work with a table valued parameter, or a TVP in short. Before using a TVP you need to prepare a table type, which is a table definition that you store as an object in the database and later use as a type for table variables and TVPs. Run the following code to create the type dbo.OrderLines for this purpose:

```
DROP TYPE IF EXISTS dbo.OrderLines;
GO
CREATE TYPE dbo.OrderLines AS TABLE
(
  productid INT           NOT NULL PRIMARY KEY,
  unitprice MONEY         NOT NULL CHECK (unitprice >= 0),
  qty       SMALLINT      NOT NULL CHECK (qty > 0),
  discount  NUMERIC(4, 3) NOT NULL CHECK (discount BETWEEN 0 AND 1)
);
```

Run the following code to create the stored procedure dbo.AddOrder:

```
CREATE OR ALTER PROC dbo.AddOrder
  @custid         AS INT,
  @empid          AS INT,
  @orderdate      AS DATE,
  @requireddate   AS DATE,
  @shippeddate    AS DATE,
  @shipperid      AS INT,
  @freight        AS MONEY,
  @shipname       AS NVARCHAR(40),
  @shipaddress    AS NVARCHAR(60),
  @shipcity       AS NVARCHAR(15),
  @shipregion     AS NVARCHAR(15),
  @shippostalcode AS NVARCHAR(10),
  @shipcountry    AS NVARCHAR(15),
  @OrderLines     AS dbo.OrderLines READONLY,
  @neworderid     AS INT OUT
AS
```

```
SET XACT_ABORT, NOCOUNT ON;

BEGIN TRAN;

-- add order header
INSERT INTO Sales.Orders
    (custid, empid, orderdate, requireddate, shippeddate,
     shipperid, freight, shipname, shipaddress, shipcity,
     shippostalcode, shipcountry)
  VALUES
    (@custid, @empid, @orderdate, @requireddate, @shippeddate,
     @shipperid, @freight, @shipname, @shipaddress, @shipcity,
     @shippostalcode, @shipcountry);

SET @neworderid = SCOPE_IDENTITY();

-- add order lines
INSERT INTO Sales.OrderDetails(orderid, productid, unitprice, qty, discount)
  SELECT @neworderid, productid, unitprice, qty, discount
  FROM @OrderLines;

COMMIT TRAN;
```

The first 13 parameters contain the order header's attributes. The 14th parameter is a TVP named @OrderLines and is of the OrderLines table type. Notice that it has to be marked as READONLY, meaning that you cannot apply changes to its contents. In our case, it is sufficient as a read-only parameter. The 15th parameter is an output parameter through which you return the newly generated order ID to the caller.

The procedure starts by setting XACT_ABORT and NOCOUNT to ON as suggested earlier. You want to make sure that if any errors occur, you never leave the transaction open. The code then performs the addition of the order header and order lines in an explicit transaction. What's currently missing is error handling code, but I take care of this in the section Error handling with TRY-CATCH.

To execute the stored procedure from T-SQL, you first need to declare a table variable of the OrderLines table type and populate it with rows. You then assign the table variable to the TVP when you call the stored procedure. You also prepare a variable to collect the newly generated order ID from the output parameter. Here's an example for executing the stored procedure:

```
DECLARE @MyOrderLines AS dbo.OrderLines, @myneworderid AS INT;

INSERT INTO @MyOrderLines(productid, unitprice, qty, discount)
  VALUES(1, 50.00, 5, 0.000),
        (2, 50.00, 5, 0.000),
        (3, 50.00, 5, 0.000);

EXEC dbo.AddOrder
  @custid         = 5,
  @empid          = 5,
  @orderdate      = '20170212',
  @requireddate   = '20170301',
```

```
  @shippeddate    = '20170216',
  @shipperid      = 2,
  @freight        = 50.00,
  @shipname       = N'Shipper 2',
  @shipaddress    = N'Address BBB',
  @shipcity       = N'City BBB',
  @shipregion     = N'Region BBB',
  @shippostalcode = N'22222',
  @shipcountry    = N'Country BBB',
  @OrderLines     = @MyOrderLines,
  @neworderid     = @myneworderid OUT;

PRINT 'Added new order with order ID '
  + CAST(@myneworderid AS VARCHAR(10)) + '.';
```

This code generates the following output:

```
Added new order with order ID 11082.
```

Run the following code to query the data for the new order:

```
SELECT orderid, orderdate, custid, empid
FROM Sales.Orders
WHERE orderid = 11082;

SELECT orderid, productid, qty
FROM Sales.OrderDetails
WHERE orderid = 11082;
```

This code generates the following output:

```
orderid     orderdate  custid      empid
----------- ---------- ----------- -----------
11082       2017-02-12 5           5

orderid     productid   qty
----------- ----------- ------
11082       1           5
11082       2           5
11082       3           5
```

Next, run the following code to execute the stored procedure from an explicit transaction, and then roll the transaction back:

```
BEGIN TRAN;

DECLARE @MyOrderLines AS dbo.OrderLines, @myneworderid AS INT;

INSERT INTO @MyOrderLines(productid, unitprice, qty, discount)
  VALUES(1, 60.00, 6, 0.000),
        (2, 60.00, 6, 0.000),
        (3, 60.00, 6, 0.000);

EXEC dbo.AddOrder
  @custid         = 6,
  @empid          = 6,
```

```
  @orderdate      = '20170212',
  @requireddate   = '20170301',
  @shippeddate    = '20170216',
  @shipperid      = 3,
  @freight        = 60.00,
  @shipname       = N'Shipper 3',
  @shipaddress    = N'Address CCC',
  @shipcity       = N'City CCC',
  @shipregion     = N'Region CCC',
  @shippostalcode = N'33333',
  @shipcountry    = N'Country CCC',
  @OrderLines     = @MyOrderLines,
  @neworderid     = @mynewOrderid OUT;

PRINT 'Added new order with order ID '
  + CAST(@mynewOrderid AS VARCHAR(10)) + '.';

ROLLBACK TRAN;
```

This code generates the following output:

```
Added new order with order ID 11083.
```

The COMMIT TRAN statement within the stored procedure doesn't truly commit the transaction because it doesn't reduce @@TRANCOUNT to zero. Because the outer batch issues a ROLLBACK TRAN statement, the work that added the header and lines of order 11083 was undone. Query the tables to look for the data of the new order:

```
SELECT orderid, orderdate, custid, empid
FROM Sales.Orders
WHERE orderid = 11083;

SELECT orderid, productid, qty
FROM Sales.OrderDetails
WHERE orderid = 11083;
```

This code generates the following output:

```
orderid     orderdate  custid      empid
----------- ---------- ----------- -----------

orderid     productid   qty
----------- ----------- ------
```

Note that if you insist on actually committing the transaction in your stored procedure irrespective of whether you started it or not, you can use a loop that keeps committing until @@TRANCOUNT drops to zero, like so:

```
WHILE @@TRANCOUNT > 0
  COMMIT TRAN;
```

This isn't considered a very good practice because someone who started a transaction and then called your procedure also expects to be able to control whether to commit it or roll it back. Also, if there's a mismatch between the @@TRANCOUNT values when the procedure

starts and when it finishes, SQL Server generates error 266 indicating the mismatch. Though this error doesn't terminate an open transaction and doesn't abort the execution of the code even if XACT_ABORT is set to ON.

To demonstrate this I use a stored procedure called dbo.Proc1 that uses an explicit transaction to create a table called dbo.DoIExist, with a loop that keeps committing until @@TRANCOUNT drops to zero. Run the following code to create the stored procedure:

```
CREATE OR ALTER PROC dbo.Proc1
AS

SET XACT_ABORT, NOCOUNT ON;

BEGIN TRAN;

PRINT 'In transaction in proc. @@TRANCOUNT is '
  + CAST(@@TRANCOUNT AS VARCHAR(10)) + '.';

CREATE TABLE dbo.DoIExist(col1 int);

WHILE @@TRANCOUNT > 0
  COMMIT TRAN;

PRINT 'Still in proc. @@TRANCOUNT is '
  + CAST(@@TRANCOUNT AS VARCHAR(10)) + '.';
GO
```

Use the following code to test the procedure without first opening a transaction in the outer batch:

```
DROP TABLE IF EXISTS dbo.DoIExist;

EXEC dbo.Proc1;

IF OBJECT_ID('dbo.DoIExist') IS NOT NULL
  PRINT 'DoIExist exists.'
ELSE
  PRINT 'DoIExist does not exist.';

PRINT 'Still in batch. @@TRANCOUNT is '
  + CAST(@@TRANCOUNT AS VARCHAR(10)) + '.';
```

This code generates the following output, as expected:

```
In transaction in proc. @@TRANCOUNT is 1.
Still in proc. @@TRANCOUNT is 0.
DoIExist exists.
Still in batch. @@TRANCOUNT is 0.
```

Now execute the stored procedure from an outer transaction that you wish to roll back:

```
DROP TABLE IF EXISTS dbo.DoIExist;

BEGIN TRAN;
```

```
EXEC dbo.Proc1;

PRINT 'Still in batch. @@TRANCOUNT is '
  + CAST(@@TRANCOUNT AS VARCHAR(10)) + '.';

IF @@TRANCOUNT > 0
  ROLLBACK TRAN;

IF OBJECT_ID('dbo.DoIExist') IS NOT NULL
  PRINT 'DoIExist exists.'
ELSE
  PRINT 'DoIExist does not exist.';
```

This code generates the following output:

```
In transaction in proc. @@TRANCOUNT is 2.
Still in proc. @@TRANCOUNT is 0.
Msg 266, Level 16, State 2, Procedure dbo.Proc1, Line 0 [Batch Start Line 436]
Transaction count after EXECUTE indicates a mismatching number of BEGIN and COMMIT
statements. Previous count = 1, current count = 0.
Still in batch. @@TRANCOUNT is 0.
DoIExist exists.
```

The stored procedure actually committed the transaction even though it wasn't the one that opened it. You get an error indicating the mismatch in the @@TRANCOUNT values when entering and leaving the procedure, but this error doesn't stop the execution of the code and doesn't cause the transaction to roll back. The table creation gets committed and the outer batch's ROLLBACK TRAN statement doesn't get the chance to execute.

Working with named transactions, savepoints, and markers

T-SQL supports indicating a name as part of the ROLLBACK TRAN[SACTION] statement, as in:

```
ROLLBACK TRAN SomeName;
```

The name has to have a binary match either with an outermost transaction name that you assigned earlier with the statement BEGIN TRAN[SACTION] <name>, or a savepoint name that you assigned earlier with the statement SAVE TRAN[SACTION] <name>.

The idea behind rolling back to a transaction name is to verify that you're explicitly rolling back the outermost transaction. If that's not the case, you get an error. As an example, run the following code:

```
SET XACT_ABORT OFF;
BEGIN TRAN OutermostTran;
BEGIN TRAN InnerTran1;
BEGIN TRAN InnerTran2;
ROLLBACK TRAN OutermostTran;
PRINT '@@TRANCOUNT is ' + CAST(@@TRANCOUNT AS VARCHAR(10)) + '.';
```

Because there's a match between the names in the ROLLBACK TRAN statement and the outermost BEGIN TRAN statement, the outermost transaction is rolled back successfully. This code generates the following output:

```
@@TRANCOUNT is 0.
```

As another example, run the following code, this time specifying an inner transaction name in the ROLLBACK TRAN statement:

```
SET XACT_ABORT OFF;
BEGIN TRAN OutermostTran;
BEGIN TRAN InnerTran1;
BEGIN TRAN InnerTran2;
ROLLBACK TRAN InnerTran1;
PRINT '@@TRANCOUNT is ' + CAST(@@TRANCOUNT AS VARCHAR(10)) + '.';
GO
ROLLBACK TRAN;
```

The attempt to roll the transaction back fails, the transaction remains open, and the code continues execution. This code generates the following output:

```
Msg 6401, Level 16, State 1, Line 472
Cannot roll back InnerTran1. No transaction or savepoint of that name was found.
@@TRANCOUNT is 3.
```

Keep in mind, though, that the above examples are executed with XACT_ABORT set to OFF, as is the case by default. If you follow best practices and set this option to ON, in the second example the error—not the ROLLBACK TRAN statement itself—causes the open transaction to roll back and the execution of the code to abort.

If you issue a ROLLBACK TRAN statement without a name, it behaves as usual irrespective of whether you named transactions or not; namely, it rolls the entire transaction back.

You can also specify a transaction name in the COMMIT TRAN[SACTION] statement, as in COMMIT TRAN SomeName, but with this command the name is simply ignored.

Naming transactions is a very uncommonly used practice. This book covers it mainly for the small chance that it would appear in the exam.

T-SQL also supports a concept called savepoints. A *savepoint* is a marker within an open transaction that you can later roll back to, undoing only the changes that took place since the savepoint, and not the rest of the changes that took place in the transaction. When you roll back to a save point, the transaction remains open and the code continues execution. You mark a savepoint with the statement SAVE TRAN[ACTION] <savepoint name>, and roll back to a savepoint using the statement ROLLBACK TRAN[SACTION] <savepoint name>. You must be in an open transaction to mark a savepoint; otherwise you get an error. You are allowed to mark multiple savepoints within the same transaction. You can gradually roll back to different savepoints, but in such a case you have to keep going backwards; otherwise you get an error. For example, if you marked save points S1, S2, S3 in this order, you can later first roll back to S3 and afterwards to S1, but not the other way around.

The following example demonstrates using savepoints:

```
SET XACT_ABORT, NOCOUNT ON;
DROP TABLE IF EXISTS dbo.T1;
GO
CREATE TABLE dbo.T1(col1 VARCHAR(10));
```

```
GO

BEGIN TRAN;

SAVE TRAN S1;
INSERT INTO dbo.T1(col1) VALUES('S1');

SAVE TRAN S2;
INSERT INTO dbo.T1(col1) VALUES('S2');

SAVE TRAN S3;
INSERT INTO dbo.T1(col1) VALUES('S3');

ROLLBACK TRAN S3;

ROLLBACK TRAN S2;

SAVE TRAN S4;
INSERT INTO dbo.T1(col1) VALUES('S4');

COMMIT TRAN;

SELECT col1 FROM dbo.T1;
GO
DROP TABLE IF EXISTS dbo.T1;
```

This code generates the following output:

```
col1
----------
S1
S4
```

When the code issued the statement ROLLBACK TRAN S3, the changes that took place between S3 and that statement were undone (the insertion of the value 'S3' to the table). When the code then issued the statement ROLLBACK TRAN S2, the changes that took place between S2 and S3 were undone (the insertion of the value 'S2' to the table). So, both the insertion of the value 'S1', which happened before S2, and the insertion of the value 'S4', which happened after the last rollback to a savepoint took place, persisted.

The BEGIN TRAN[SACTION] statement also supports an option called WITH MARK <marker name>, as in BEGIN TRAN MyTran WITH MARK 'My Mark'. This option defines a marker name in the transaction log. Later on, if you apply a restore from a log backup, you can specify the option STOPATMARK <marker name> to recover the changes only until the indicated marked transaction, inclusive, or STOPBEFOREMARK <marker name> to exclude the marked transaction.

Error handling with TRY-CATCH

Robust programming solutions always include code to handle errors. You want to make sure that you control the outcome of both anticipated and unanticipated errors to the degree that you can. This is true with any kind of programming, and of course also specifically with T-SQL. In case of an error, you do not want to leave a database in an inconsistent state. Also, you do not want an error to result in an unterminated transaction because this causes concurrency and other problems in the system.

This section describes the error handling tools that T-SQL supports. It covers the main T-SQL error handling construct—TRY-CATCH, error functions, generating error messages with THROW and RAISERROR, and handling errors in transactions.

The TRY-CATCH construct

T-SQL uses a classic TRY-CATCH construct to handle errors, similar to what most programming languages use for error handling. You place your usual code within the TRY block, and you place any error handling code within the CATCH block. If there's no error in the TRY block, the CATCH block is skipped. If there is an error in the TRY block, control is passed to the first line of code within the CATCH block.

As the first example for using the TRY-CATCH construct I demonstrate code that adds a couple of rows to a table called T1. Run the following code to create the table T1:

```
SET NOCOUNT ON;
USE TSQLV4;

DROP TABLE IF EXISTS dbo.T1;
GO
CREATE TABLE dbo.T1
(
  keycol INT NOT NULL
    CONSTRAINT PK_T1 PRIMARY KEY,
  col1 INT NOT NULL
    CONSTRAINT CHK_T1_col1_gtzero CHECK(col1 > 0),
  col2 VARCHAR(10) NOT NULL
);
```

The following example demonstrates execution of code with no errors:

```
BEGIN TRY

  INSERT INTO dbo.T1(keycol, col1, col2)
    VALUES(1, 10, 'AAA');
  INSERT INTO dbo.T1(keycol, col1, col2)
    VALUES(2, 20, 'BBB');

  PRINT 'Got to end of TRY block.';

END TRY
BEGIN CATCH
```

```
  PRINT 'Error occurred. Entering CATCH block. Error message: ' + ERROR_MESSAGE();

END CATCH;
GO

SELECT keycol, col1, col2
FROM dbo.T1;

-- cleanup
TRUNCATE TABLE dbo.T1;
```

The code in the TRY block runs successfully and therefore the CATCH block is skipped. This code generates the following output:

```
Got to the end of the TRY block.
keycol      col1        col2
----------- ----------- ----------
1           10          AAA
2           20          BBB
```

The following example demonstrates execution of code with an error:

```
BEGIN TRY

  INSERT INTO dbo.T1(keycol, col1, col2)
    VALUES(1, 10, 'AAA');
  INSERT INTO dbo.T1(keycol, col1, col2)
    VALUES(2, -20, 'BBB');

  PRINT 'Got to the end of the TRY block.';

END TRY
BEGIN CATCH

  PRINT 'Error occurred. Entering CATCH block. Error message: ' + ERROR_MESSAGE();

END CATCH;
GO

SELECT keycol, col1, col2
FROM dbo.T1;

-- cleanup
TRUNCATE TABLE dbo.T1;
```

The first INSERT statement runs successfully and the row makes it to the table. The second INSERT statement fails due to a CHECK constraint violation when it tries to add a row with a col1 value that is not greater than zero. The second row doesn't make it to the table. The PRINT command at the end of the TRY block doesn't get to execute. Control of the code passes to the CATCH block, which in this example simply prints a message indicating that an error occurred along with the error message. This code generates the following output:

```
Error occurred. Entering CATCH block. Error message: The INSERT statement conflicted
with the CHECK constraint "CHK_T1_col1_gtzero". The conflict occurred in database
"TSQLV4", table "dbo.T1", column 'col1'.
keycol      col1        col2
----------- ----------- ----------
1           10          AAA
```

This is hardly an example for robust error handling because this code leaves the database in an inconsistent state with only one of the two rows making it into the target table. Normally you would place the work in a transaction and make sure that when an error happens you roll the transaction back, but I get to this later. For now, I just wanted to demonstrate the flow of the code with the TRY-CATCH construct when it runs successfully and when there is an error.

If an error happens within a stored procedure and is handled by a TRY-CATCH construct, as far as the caller of the stored procedure is concerned, there was no error. If an error happens in a stored procedure, but not in a TRY block, the error bubbles up in the call stack until it finds a TRY block, and if one is found, it activates the corresponding CATCH block. For example, say Proc1 calls Proc2 from within a TRY block, and Proc2 runs code without using a TRY-CATCH construct. An error in Proc2 results in the CATCH block of Proc1 being activated. If no TRY-CATCH construct is found along the way, the caller who initiated the code ends up getting the error.

EXAM TIP

For cases where code within the CATCH bock can potentially fail, you are allowed to nest a TRY-CATCH construct within the CATCH block, and this way handle such errors. If an error happens in a CATCH block, but not in a nested TRY-CATCH construct, the error behaves as if it didn't happen in a TRY block; namely, it bubbles up.

To demonstrate nesting of TRY-CATCH constructs I use a procedure called AddRowToT1 and a table called ErrorLog where the procedure is supposed to log errors.

Run the following code to create the table ErrorLog:

```
DROP TABLE IF EXISTS dbo.ErrorLog;
GO
CREATE TABLE dbo.ErrorLog
(
  id INT NOT NULL IDENTITY
    CONSTRAINT PK_ErrorLog PRIMARY KEY,
  dt DATETIME2 NOT NULL DEFAULT(SYSDATETIME()),
  loginname NVARCHAR(128) NOT NULL DEFAULT(SUSER_SNAME()),
  errormessage NVARCHAR(4000) NOT NULL
);
```

The table definition uses default constraints to record the time of the error and the executing login.

Run the following code to create the stored procedure AddRowToT1:

```
CREATE OR ALTER PROC dbo.AddRowToT1
  @keycol INT,
  @col1   INT,
  @col2   VARCHAR(10)
AS

SET NOCOUNT ON;

BEGIN TRY

  INSERT INTO dbo.T1(keycol, col1, col2)
    VALUES(@keycol, @col1, @col2);

  PRINT 'Got to the end of the outer TRY block.';

END TRY
BEGIN CATCH

  PRINT 'Error occurred in outer TRY block. Entering outer CATCH block.';

  BEGIN TRY

    INSERT INTO dbo.ErrorLog(errormessage) VALUES(ERROR_MESSAGE());

    PRINT 'Got to the end of the inner TRY block.';

  END TRY
  BEGIN CATCH

    PRINT 'Error occurred in inner TRY block. Entering inner CATCH block. Error message:
' + ERROR_MESSAGE();

  END CATCH;

END CATCH;
GO
```

As you can see, the stored procedure uses an outer TRY block to add a row to T1 with the values received as input parameters. If all goes well, the outer TRY block prints a message indicating that it got to the end. Note that the PRINT statements are used here only for illustration purposes to show which parts of the code got executed; normally, you would not include those in production code. In case of an error, the outer CATCH block is activated. The code in the outer CATCH block prints that it got there, and then activates an inner TRY-CATCH construct. The inner TRY block tries to write the error information into the table ErrorLog. If all goes well, the code indicates that it got to the end of the inner TRY block. If there's an error in the inner TRY block, the inner CATCH block is activated. The inner CATCH block prints a message indicating the code got there and the error message.

Use the following code to test the procedure for the first time:

```
EXEC dbo.AddRowToT1
  @keycol = 1,
  @col1 = 10,
  @col2 = 'AAA';
GO

SELECT keycol, col1, col2 FROM dbo.T1;
SELECT id, dt, loginname, errormessage FROM dbo.ErrorLog;

-- cleanup
TRUNCATE TABLE dbo.T1;
TRUNCATE TABLE dbo.ErrorLog;
```

The code runs successfully with no errors. The code generates the following output:

```
Got to the end of the outer TRY block.
keycol      col1        col2
----------- ----------- ----------
1           10          AAA

id   dt  loginname  errormessage
---- --- ---------- -------------
```

As you can see, the outer TRY block executed to completion. The code added a new row into the table T1. The outer CATCH block was skipped, and no row was added to the table ErrorLog.

Run the following code in attempt to add a row with an invalid col1 value:

```
EXEC dbo.AddRowToT1
  @keycol = 1,
  @col1 = -10,
  @col2 = 'BBB';
GO

SELECT keycol, col1, col2 FROM dbo.T1;
SELECT id, dt, loginname, errormessage FROM dbo.ErrorLog;

-- cleanup
TRUNCATE TABLE dbo.T1;
TRUNCATE TABLE dbo.ErrorLog;
```

This time there's an error when the outer TRY block attempts to add the row into T1, therefore the outer CATCH block is activated. The outer CATCH block uses an inner TRY-CATCH construct to add a row with the error information into the table ErrorLog. The inner TRY block succeeds in this task and therefore the inner CATCH block is skipped. This code generates the following output (formatted for clarity):

```
Error occurred in outer TRY block. Entering outer CATCH block.
Got to the end of the inner TRY block.
keycol      col1        col2
----------- ----------- ----------
```

```
id          dt                          loginname
----------- --------------------------- -------------
1           2017-01-24 09:44:24.1014963 <your_login>

errormessage
-------------------------------------------------------------------------------------------
-------------------------------------------------------------------
The INSERT statement conflicted with the CHECK constraint "CHK_T1_col1_gtzero". The
conflict occurred in database "TSQLV4", table "dbo.T1", column 'col1'.
```

The new row didn't make it into T1. A row with the error info was written into ErrorLog.

To test an example for an error in the inner CATCH block, open two new connections (I refer to them as connection 1 and connection 2). Run the following code in connection 1 to obtain an exclusive table lock on the table ErrorLog within a transaction that the code leaves open:

```
BEGIN TRAN;

SELECT TOP (0) * FROM dbo.ErrorLog WITH (TABLOCKX);
```

Next, run the following code to first set the session's lock expiration time out to zero (no wait), and then to attempt to add an invalid row:

```
SET LOCK_TIMEOUT 0;

EXEC dbo.AddRowToT1
  @keycol = 1,
  @col1 = -10,
  @col2 = 'BBB';
```

The outer TRY block fails when trying to add the invalid row, activating the outer CATCH block. The outer CATCH block activates the inner TRY-CATCH construct. The inner TRY block fails when trying to add a row to the table ErrorLog because it cannot obtain a lock. Because you set the lock expiration time out to zero, the inner CATCH block is immediately activated, printing the inner error information. This code generates the following output:

```
Error occurred in outer TRY block. Entering outer CATCH block.
Error occurred in inner TRY block. Entering inner CATCH block. Error message: Lock
request time out period exceeded.
```

Run the following code in connection 1 to commit the open transaction and release the lock on ErrorLog:

```
COMMIT TRAN;
```

Run the following code in connection 2 to query the tables, clean them up, and to set the lock expiration time out back to infinity:

```
SELECT keycol, col1, col2 FROM dbo.T1;
SELECT id, dt, loginname, errormessage FROM dbo.ErrorLog;

-- cleanup
TRUNCATE TABLE dbo.T1;
```

```
TRUNCATE TABLE dbo.ErrorLog;
SET LOCK_TIMEOUT -1;
```

This code generates the following output showing that both tables are empty:

```
keycol      col1        col2
----------- ----------- ----------

id   dt  loginname  errormessage
---- --- ---------- -------------
```

Close connection 2 and return to connection 1 before continuing.

An important limitation of TRY-CATCH that you want to make sure that you're aware of, and that you plan for, is that it cannot catch compilation errors, such as referring to objects and columns that don't exist, in the same scope. However, you can catch such errors in an outer scope.

As an example, consider the following stored procedure called InnerProc:

```
CREATE OR ALTER PROC dbo.InnerProc
AS

BEGIN TRY
  SELECT nosuchcolumn FROM dbo.NoSuchTable;
END TRY
BEGIN CATCH
  PRINT 'In CATCH block of InnerProc.';
END CATCH;
GO
```

The procedure tries to query a table that doesn't exist.

Run the following code to execute InnerProc:

```
EXEC dbo.InnerProc;
```

Because the error is a compilation error, it isn't caught by the inner CATCH block. This execution generates the following error:

```
Msg 208, Level 16, State 1, Procedure dbo.InnerProc, Line 5 [Batch Start Line 703]
Invalid object name 'dbo.NoSuchTable'.
```

Use the following code to create a stored procedure called OuterProc that invokes InnerProc within a TRY block:

```
CREATE OR ALTER PROC dbo.OuterProc
AS

BEGIN TRY
  EXEC dbo.InnerProc;
END TRY
BEGIN CATCH
  PRINT 'In CATCH block of OuterProc.';
END CATCH;
GO
```

Use the following code to test OuterProc:

```
EXEC dbo.OuterProc;
```

This time the CATCH block of OuterProc is activated, and the execution generates the following output:

```
In CATCH block of OuterProc.
```

Error functions

T-SQL supports six error functions that provide information about the error, and that you can query in the CATCH block. Following are the functions and their descriptions:

- ERROR_NUMBER() provides the error number.
- ERROR_MESSAGE() provides the error message.
- ERROR_SEVERITY() provides the error severity. You can catch errors with severity 11 to 19. Errors with severity 20 and up are so severe that when they happen, SQL Server terminates your connection. Any error handling code that you may have doesn't really have a chance to run. Messages with severity 0 to 9 are considered informational and are always passed to the client; they're not accessible to SQL Server. Messages with severity 10 are also informational; for compatibility reasons, SQL Server converts those to severity 0 internally. For more information about error severities, see the topic "Database Engine Error Severities" at *https://msdn.microsoft.com/en-us/library/ms164086.aspx*.
- ERROR_STATE() provides the error state. The error state is an integer in the range 1 to 255. It can be used for different custom purposes such as indicating where the error originated in cases where the error can happen in different places in the SQL Server engine's code.
- ERROR_LINE() provides the line number where the error happened.
- ERROR_PROCEDURE() provides the name of the stored procedure where the error happened. If the error did not happen in a stored procedure, this function returns a NULL.

There are a number of important things you want to keep in mind when using these functions:

- If you invoke these functions not within a CATCH block, they all return NULLs.
- If you nest TRY-CATCH constructs, whether directly or indirectly when you call one procedure from another, these functions return error information about the innermost error.
- Some failures generate a chain of errors. These functions only return information about the last error in the chain.

Conveniently, you are allowed to invoke these functions from a stored procedure for reusability. Still, the ERROR_PROCEDURE function returns the name of the stored procedure where the error happened and not the name of the procedure where you invoked the function, if

different. To demonstrate this, first run the following code to create the stored procedure PrintErrorInfo, which simply prints the values of the error functions:

```
CREATE OR ALTER PROC dbo.PrintErrorInfo
AS

PRINT 'Error Number  : ' + CAST(ERROR_NUMBER() AS VARCHAR(10));
PRINT 'Error Message : ' + ERROR_MESSAGE();
PRINT 'Error Severity: ' + CAST(ERROR_SEVERITY() AS VARCHAR(10));
PRINT 'Error State   : ' + CAST(ERROR_STATE() AS VARCHAR(10));
PRINT 'Error Line    : ' + CAST(ERROR_LINE() AS VARCHAR(10));
PRINT 'Error Proc    : ' + COALESCE(ERROR_PROCEDURE(), 'Not within proc');
GO
```

Next, run the following code to alter the AddRowToT1 procedure to run the PrintErrorInfo procedure in the CATCH block when an error happens in the attempt to insert the row into T1 in the TRY block:

```
CREATE OR ALTER PROC dbo.AddRowToT1 -- proc name
  @keycol INT,
  @col1   INT,
  @col2   VARCHAR(10)
AS

SET NOCOUNT ON;

BEGIN TRY

  INSERT INTO dbo.T1(keycol, col1, col2) -- line 11
    VALUES(@keycol, @col1, @col2);

END TRY
BEGIN CATCH

  EXEC dbo.PrintErrorInfo;

END CATCH;
GO
```

Run the following code to execute the AddRowToT1 procedure with an invalid value for col1:

```
EXEC dbo.AddRowToT1
  @keycol = 1,
  @col1 = -10,
  @col2 = 'AAA';
```

This code generates the following output:

```
Error Number  : 547
Error Message : The INSERT statement conflicted with the CHECK constraint "CHK_T1_col1_
gtzero". The conflict occurred in database "TSQLV4", table "dbo.T1", column 'col1'.
Error Severity: 16
Error State   : 0
Error Line    : 11
Error Proc    : dbo.AddRowToT1
```

Notice that the stored procedure name where the error actually happened was captured correctly. Also notice that the line number reported is the line number where the error happened within the stored procedure.

The THROW and RAISERROR commands

The THROW and RAISERROR (yes there's only one E in there) commands allow you to raise an error. Both allow you to raise a user defined error, but only THROW allows you to re-throw an original error that was caught by a TRY-CATCH construct.

THE THROW COMMAND

The THROW command has two supported syntaxes. One is without parameters, with which you use the command in a CATCH block to re-throw the error that originally activated that CATCH block. The re-thrown error behaves like the original one, with the same error number, severity, and state. Also, if the failure normally generates a chain of error messages, the re-thrown error does too. Irrespective of whether the original error is a batch-aborting one or not, the THROW command aborts the batch and bubbles up, unless you call it from a nested TRY-CATCH construct, in which case it activates the inner CATCH block.

The following procedure demonstrates using THROW without parameters:

```
CREATE OR ALTER PROC dbo.Divide
  @dividend AS INT,
  @divisor  AS INT
AS

SET NOCOUNT ON;

BEGIN TRY

  SELECT @dividend / @divisor AS quotient, @dividend % @divisor AS remainder;

END TRY
BEGIN CATCH

  PRINT 'Error occurred when trying to compute the division '
    + CAST(@dividend AS VARCHAR(11)) + ' / ' + CAST(@divisor AS VARCHAR(11)) + '.';

  THROW;

  PRINT 'This doesn''nt execute.';

END CATCH;
GO
```

The procedure Divide uses a TRY block to invoke a query that computes the quotient and remainder of integer division applied to the two input parameters. If all goes well, the query returns the result of the calculations. In case of an error, the CATCH block is activated. The CATCH block first prints a message indicating that an error occurred with the attempted calculation, and then re-throws the original error assuming that you want it to bubble up to

the caller. To show that the THROW command aborts the batch, the CATCH block has a PRINT statement after the THROW command, but this statement never gets to execute.

Use the following code to first test the procedure with valid inputs:

```
EXEC dbo.Divide @dividend = 11, @divisor = 2;
```

The procedure executes successfully, generating the following output:

```
quotient    remainder
----------- -----------
5           1
```

Execute the procedure again, but this time with zero as the divisor:

```
EXEC dbo.Divide @dividend = 11, @divisor = 0;
```

This code generates the following output:

```
quotient    remainder
----------- -----------

Error occurred when trying to compute the division 11 / 0.
Msg 8134, Level 16, State 1, Procedure dbo.Divide, Line 8 [Batch Start Line 799]
Divide by zero error encountered.
```

A *divide by zero* error happened. The CATCH block was activated. The CATCH block printed a message saying "Error occurred when trying to compute the division 11 / 0." The CATCH block then re-threw the error, at which point SQL Server aborted the batch and passed the error to the caller. Then next line of code in the CATCH block after the THROW command didn't get to execute.

You should be aware of potential parsing ambiguity with code that has an unterminated statement preceding the THROW command. As an example, run the following code to alter the definition of the procedure Divide, and execute it with zero as the divisor:

```
CREATE OR ALTER PROC dbo.Divide
  @dividend AS INT,
  @divisor  AS INT
AS

SET NOCOUNT ON;

BEGIN TRY

  SELECT @dividend / @divisor AS quotient, @dividend % @divisor AS remainder;

END TRY
BEGIN CATCH

  SELECT 'What comes next is an alias'

  THROW;

END CATCH;
GO
```

```
EXEC dbo.Divide @dividend = 11, @divisor = 0;
GO
```

This code generates the following output:

```
quotient    remainder
----------- -----------

THROW
--------------------------
What comes next is an alias
```

Notice that SQL Server didn't re-throw the error because the parser assumed that the code uses THROW as a column alias for the expression in the preceding unterminated query, as if the intended query was:

```
SELECT 'What comes next is an alias' THROW;
```

As another example for ambiguity, run the following code to alter the procedure definition again, and execute it:

```
CREATE OR ALTER PROC dbo.Divide
  @dividend AS INT,
  @divisor  AS INT
AS

SET NOCOUNT ON;

BEGIN TRY

  SELECT @dividend / @divisor AS quotient, @dividend % @divisor AS remainder;

END TRY
BEGIN CATCH

  IF @@TRANCOUNT > 0 ROLLBACK TRAN

  THROW;

END CATCH;
GO

EXEC dbo.Divide @dividend = 11, @divisor = 0;
```

Here your goal in the CATCH block is to roll back an active transaction if one was opened by calling code before you re-throw the error. However, because the preceding ROLLBACK TRAN statement isn't terminated, the parser thinks that THROW is a transaction or savepoint name, rather than being a command of its own, as if the statement was ROLLBACK TRAN THROW. In the above execution, there's no transaction open, so the ROLLBACK TRAN THROW statement doesn't get to execute, and the problem silently goes unnoticed. The error is not re-thrown. This code generates the following output:

```
quotient    remainder
----------- -----------
```

Next, execute the code from within a user transaction:

```
BEGIN TRAN;

EXEC dbo.Divide @dividend = 11, @divisor = 0;

PRINT '@@TRANCOUNT is ' + CAST(@@TRANCOUNT AS VARCHAR(10)) + '.';
```

This time the CATCH block tries to execute the command ROLLBACK TRAN THROW, but because it cannot find a transaction or savepoint name called THROW, it generates an error indicating as much, and does not re-throw the original divide by zero error. Worse, it leaves the transaction open! This code generates the following output:

```
quotient    remainder
----------- -----------

Msg 6401, Level 16, State 1, Procedure dbo.Divide, Line 15 [Batch Start Line 58]
Cannot roll back THROW. No transaction or savepoint of that name was found.
@@TRANCOUNT is 1.
```

Run the following code to roll back the currently open transaction:

```
ROLLBACK TRAN;
```

Clearly, this ambiguity can get you into quite a lot of trouble. The official documentation for THROW states that a statement preceding the command must be terminated. If indeed you terminate the preceding statement, there is no ambiguity. If you have a coding policy to terminate all statements, you avoid ambiguity anyway. The reality though is that most people simply don't follow the best practice to terminate all statements, so most T-SQL code out there isn't properly terminated. To be on the safe side, some people developed a practice to always precede the THROW command with a terminator, as if the command was actually ;THROW. This way you don't need to worry about whether the preceding statement is, or might be, terminated.

A very similar ambiguity exists with the WITH clause which can be used to define a CTE as well as for other purposes like specifying a table hint in a query. Therefore, the documentation indicates that you must terminate a statement preceding a WITH clause that defines a CTE. Similar to the unofficial practice to prefix THROW with a terminator, some people also regularly prefix a WITH clause that defines a CTE with a terminator, as in ;WITH.

The second syntax for the THROW command is with parameters. You use this syntax to throw a user-defined error. You can use it both inside and outside a CATCH block. The general form of THROW with parameters is:

```
THROW errornumber, message, state;
```

All three parameters can be either constants or variables.

Here's an example for using THROW with constant parameters:

```
THROW 50000, 'This is a user-defined error.', 1;
```

This code generates the following output:

```
Msg 50000, Level 16, State 1, Line 865
This is a user-defined error.
```

The first parameter is a user specified integer error number that must be greater than or equal to 50000. The error number does not need to be recorded anywhere previously.

The second parameter is a user specified error message. The THROW command does not support parameter markers in the message directly. If you want to embed the percent sign (%) as part of the message, you need to escape it by indicating two percent signs (%%). If you want to embed input values as part of the message, you need to construct the message ahead in a variable by applying string concatenation, or by using the FORMATMESSAGE function, which supports using parameter markers. I demonstrate this shortly.

The third parameter is a user specified integer state in the range 1 to 255. You usually use this value to provide custom information like which place in the code the error originated from.

The THROW command does not support specifying a severity. It always generates an error with severity 16.

Here's an example of using a variable for the error message parameter:

```
DECLARE @msg AS NVARCHAR(2048) =
  'This is a user-define error that occurred on ' + CONVERT(CHAR(10), SYSDATETIME(),
121) + '.';

THROW 50000, @msg, 1;
```

This code generates the following output, with the date reflecting your execution date, of course:

```
Msg 50000, Level 16, State 1, Line 872
This is a user-define error that occurred on 2017-02-12.
```

As an alternative to constructing the message with string concatenation techniques, you can use the FORMATMESSAGE function, which allows you to embed in the message text parameter markers like the ones used by the printf function in C, like so:

```
DECLARE @msg AS NVARCHAR(2048) =
  FORMATMESSAGE('This is a user-define error that occurred on %s.',
    CONVERT(CHAR(10), SYSDATETIME(), 121));

THROW 50000, @msg, 1;
```

Commonly used markers are %s for a string and %d for an integer.

In the example above the first parameter to FORMATMESSAGE is the message text with the parameter markers, followed by values for the parameter markers. You can use an alternative syntax where the first parameter is an ID of a message stored in the sys.messages table. You can store user defined messages in sys.messages using the sp_addmessage stored procedure. See the official documentation for details at *https://msdn.microsoft.com/en-us/library/ms186788.aspx*.

If you execute the THROW command with parameters outside a TRY-CATCH construct, it always aborts the batch. If you execute it in a TRY block, it causes the corresponding CATCH block to be activated.

The following example demonstrates the batch-abort behavior:

```
THROW 50000, 'This is a user-defined error.', 1;
PRINT 'This code in the same batch doesn''t execute.';
GO
PRINT 'This code in a different batch does execute.';
```

This code generates the following output:

```
Msg 50000, Level 16, State 1, Line 884
This is a user-defined error.
This code in a different batch does execute.
```

Notice that the PRINT statement that appears right after the THROW command in the same batch doesn't get to execute, but the subsequent PRINT statement in a separate batch does.

When XACT_ABORT is off and you execute THROW while in an open transaction, the transaction remains open and committable.

When XACT_ABORT is on and you execute THROW while in an open transaction, if you're not using TRY-CATCH, SQL Server aborts the transaction, and if you are using TRY-CATCH, SQL Server dooms the transaction. I explain what doomed transactions are in the next section.

The following code demonstrates calling THROW when XACT_ABORT is set to off:

```
SET XACT_ABORT OFF;

BEGIN TRAN;

THROW 50000, 'Hello from THROW.', 1;
PRINT 'This doesn''t execute.';
GO

PRINT 'New batch... @@TRANCOUNT is ' + CAST(@@TRANCOUNT AS VARCHAR(10)) + '.';
IF @@TRANCOUNT > 0
  ROLLBACK TRAN;
```

This code generates the following output:

```
Msg 50000, Level 16, State 1, Line 894
Hello from THROW.
New batch... @@TRANCOUNT is 1.
```

Notice that the execution of the THROW command terminated the batch but not the transaction.

Run the code again, this time setting XACT_ABORT to on:

```
SET XACT_ABORT ON;
```

```
BEGIN TRAN;

THROW 50000, 'Hello from THROW.', 1;
PRINT 'This doesn''t execute.';
GO

PRINT 'New batch... @@TRANCOUNT is ' + CAST(@@TRANCOUNT AS VARCHAR(10)) + '.';
IF @@TRANCOUNT > 0
  ROLLBACK TRAN;
```

This code generates the following output:

```
Msg 50000, Level 16, State 1, Line 908
Hello from THROW.
New batch... @@TRANCOUNT is 0.
```

This time the execution of THROW caused both the batch and the transaction to abort.

The THROW command doesn't support an option to request to log the error in the SQL Server error log or the Windows application log.

THE RAISERROR COMMAND

The RAISERROR command is the predecessor to the THROW command. Some still find it useful even when writing new code because it supports certain options that THROW doesn't.

The RAISERROR command has the following syntax:

```
RAISERROR ( { message | messageid }, severity, state [, arguments for parameter markers
] ) [ WITH options ];
```

The parameters can be either constants or variables.

Here's an example for using the RAISERROR command:

```
RAISERROR( 'This is a user-define error with a string parameter %s and an integer
parameter %d.', 16, 1, 'ABC', 123 );
```

This code generates the following output:

```
Msg 50000, Level 16, State 1, Line 926
This is a user-define error with a string parameter ABC and an integer parameter 123.
```

The first parameter can be a message text, with optional parameter markers using C-like printf syntax (details can be found at *https://msdn.microsoft.com/en-us/library/ms178592.aspx*), such as in the above example. In such a case, the error number is always 50000. The parameters positioned fourth and on are the values that the command uses to replace the parameter markers. In our example, ABC replaces %s and 123 replaces %d. If you want to include the percent sign (%) in the message text, you need to escape it by specifying two percent signs (%%). Alternatively, the first parameter can be an ID of a user-defined message with a number greater than 50000 that you created earlier with the sp_addmessage stored procedure, and that was stored in the sys.messages table. You can also emulate system errors for error numbers that are greater than or equal to 13000.

The second parameter is the error severity. The outcome of raising an error with a certain severity is similar to the outcome of a system generated error with the same severity. Namely, messages with severities 0 to 10 are considered informational, and are always sent to the client, with severity 10 getting internally converted to 0 for compatibility reasons. With severities 0 and 10, the message prints without a header. With severities 1 to 9, the message prints with a header.

The following example demonstrates raising an error with severity 0:

```
RAISERROR( 'This is a message with severity 0.', 0, 1 );
```

This code generates the following output:

```
This is a message with severity 0.
```

The following example demonstrates raising an error with severity 1 to 9:

```
RAISERROR( 'This is a message with severity 1 to 9.', 1, 1 );
```

This code generates the following output:

```
This is a message with severity 1 to 9.
Msg 50000, Level 1, State 1
```

Errors with a severity level lower than 11 are not catchable. Errors with severities 11 to 19 are catchable. Errors with severities 20 to 25 terminate the connection. To raise an error with severity 19 and up you have to add the option WITH LOG, which logs the error in the SQL Server error log as well as the Windows application log. To use the WITH LOG option you must be a member of the sysadmin role or have the ALTER TRACE permission.

The third parameter is an integer state value in the range 1 to 255, similar to the state value you specify with the THROW command.

There are additional options that you can specify as part of the WITH clause of the RAISERROR command. The two most commonly used ones are LOG and NOWAIT. I already explained what the LOG option does and the required permissions to use it. The NOWAIT option causes SQL Server to send the message immediately to the client without waiting for its internal buffer to first fill up. Developers often use this option when running long scripts to report the progress of the code in intermediate milestones. The following example demonstrates using the NOWAIT option:

```
RAISERROR( 'First message.', 0, 1 ) WITH NOWAIT;
WAITFOR DELAY '00:00:05';
RAISERROR( 'Second message.', 0, 1 ) WITH NOWAIT;
```

Both messages are sent immediately to the client when the corresponding RAISERROR command executes. The first immediately prints when the code starts executing and the second after five seconds. Run the following code, which does not use the NOWAIT option:

```
RAISERROR( 'First message.', 0, 1 );
WAITFOR DELAY '00:00:05';
RAISERROR( 'Second message.', 0, 1 );
```

This time both messages get printed only after five seconds when the code completes executing.

The following example demonstrates raising an error with severity 20 using the WITH LOG option:

```
RAISERROR( 'This is a message with severity 20.', 20, 1 ) WITH LOG;
```

The message gets logged. Also, because the specified severity is 20, SQL Server terminates the connection, generating the following output:

```
Msg 50000, Level 20, State 1, Line 945
This is a message with severity 20.
Msg 596, Level 21, State 1, Line 944
Cannot continue the execution because the session is in the kill state.
Msg 0, Level 20, State 0, Line 944
A severe error occurred on the current command.  The results, if any, should be
discarded.
```

At this point you are disconnected; however, next time you execute a batch of T-SQL code, SSMS 2016 will restore the connection.

The THROW command does not support options similar to LOG and NOWAIT, giving RAISERROR an advantage when you need those.

Note that other than errors with severity 20 and up, which terminate the connection, if you raise an error with a severity that is lower than 20 this doesn't terminate the batch, nor does this terminate or doom the transaction, irrespective of the state of the XACT_ABORT option. This is one of the drawbacks of the RAISERROR command. The following code demonstrates this:

```
SET XACT_ABORT ON;

BEGIN TRAN;

RAISERROR( 'This is a user-defined error.', 16, 1 );
PRINT 'This code in the same batch executes. @@TRANCOUNT is ' + CAST(@@TRANCOUNT AS
VARCHAR(10)) + '.';

IF @@TRANCOUNT > 0
  ROLLBACK TRAN;
```

This code generates the following output:

```
Msg 50000, Level 16, State 1, Line 953
This is a user-defined error.
This code in the same batch executes. @@TRANCOUNT is 1.
```

As you can see, the batch continued executing after raising the error and the transaction remained open despite the fact that XACT_ABORT was on.

Since RAISERROR is a reserved keyword, unlike THROW, with the former a preceding statement doesn't have to be terminated, though it is still a good practice to do so.

EXAM TIP

If your exam includes questions involving RAISERROR and THROW, chances are that you are expected to know how to pick between the two based on the differences between them. For your convenience, Table 3-1 provides a comparison summary between the two tools. Make sure you understand and memorize all items in the table.

Table 3-1 has a comparison between THROW and RAISERROR.

TABLE 3-1 Comparison between THROW and RAISERROR

Proprty	THROW	RAISERROR
Can re-throw original system error	Yes	No
Activates CATCH block	Yes	Yes, for 10 < severity < 20
Always aborts batch when not using TRY-CATCH	Yes	No
Aborts/dooms transaction if XACT_ABORT is off	No	No
Aborts/dooms transaction if XACT_ABORT is on	Yes	No
If error number is passed, it must be defined in sys.messages	No	Yes
Supports printf parameter markers directly	No	Yes
Supports indicating severity	No	Yes
Supports WITH LOG to log error to error log and application log	No	Yes
Supports WITH NOWAIT to send messages immediately to the client	No	Yes
Preceding statement needs to be terminated	Yes	No

The next section covers handling errors that happen in transactions.

Error handling with transactions

When errors happen in transactions, there are additional considerations and complexities that you need to be aware of beyond what the previous sections already covered. Your course of action in case of an error could depend on the state of the transaction after the error. This section covers error handling with transactions.

As mentioned earlier, there are quite a lot of possible outcomes of errors in T-SQL. Some errors abort the batch and some don't. Some errors abort the transaction and some don't. Some errors even terminate the connection. You can achieve better consistency and some degree of control over the outcome of errors by turning the XACT_ABORT option to on, and by using the TRY-CATCH construct. This results in more robust solutions.

The following example demonstrates a divide by zero error, which normally doesn't abort the batch and doesn't terminate the transaction:

```
SET XACT_ABORT OFF;

BEGIN TRAN;

DECLARE @i AS INT = 10/0;
PRINT 'Batch wasn''t aborted.';
GO

PRINT '@@TRANCOUNT is ' + CAST(@@TRANCOUNT AS VARCHAR(10)) + '.';

IF @@TRANCOUNT > 0
  ROLLBACK TRAN;
```

This code generates the following output:

```
Msg 8134, Level 16, State 1, Line 967
Divide by zero error encountered.
Batch wasn't aborted.
@@TRANCOUNT is 1.
```

The following example demonstrates a conversion error, which aborts both the batch and the transaction:

```
BEGIN TRAN;

DECLARE @i AS INT = CAST('1,759' AS INT);
PRINT 'Batch wasn''t aborted.';
GO

PRINT '@@TRANCOUNT is ' + CAST(@@TRANCOUNT AS VARCHAR(10)) + '.';

IF @@TRANCOUNT > 0
  ROLLBACK TRAN;
```

This code generates the following output:

```
Msg 245, Level 16, State 1, Line 980
Conversion failed when converting the varchar value '1,759' to data type int.
@@TRANCOUNT is 0.
```

As you can see, it's not like a conversion error is considered more severe than a divide by zero error, at least as far as the formal severity level is concerned. But the two errors have very different outcomes. If you set the XACT_ABORT option to on, you get a more consistent behavior in the sense that most errors that normally don't terminate the transaction and/or batch, with this option turned on terminate the batch and rollback the transaction.

To demonstrate this, run the example with the divide by zero error again, after turning on the XACT_ABORT option:

```
SET XACT_ABORT ON;

BEGIN TRAN;
```

```
DECLARE @i AS INT = 10/0;
PRINT 'Batch wasn''t aborted.';
GO

PRINT '@@TRANCOUNT is ' + CAST(@@TRANCOUNT AS VARCHAR(10)) + '.';

IF @@TRANCOUNT > 0
  ROLLBACK TRAN;
```

This code generates the following output:

```
Msg 8134, Level 16, State 1, Line 995
Divide by zero error encountered.
@@TRANCOUNT is 0.
```

This time the error caused both the batch and the transaction to abort.

As you've seen, when you're not using TRY-CATCH, the transaction state after an error can be either open or not, and you can check the state using the @@TRANCOUNT. If it is open, you can determine whether you want to commit it or roll it back. The problem with cases where the transaction is aborted is that any changes done by the transaction are lost. What if as part of your error handling logic you wish to investigate data that was created by the transaction before you eventually roll it back. To this end, errors that normally cause the transaction to abort when you're not using TRY-CATCH, cause the transaction to enter a special *doomed* state (also known as *failed*, and *uncommittable* state). Under this state, you're not allowed to change data other than in table variables, but you are allowed to read data. You're not allowed to commit a doomed transaction, rather you have to eventually roll it back. You're not allowed to roll the transaction back to a savepoint, rather it has to be a full transaction rollback.

So when using TRY-CATCH, there could be three possible outcomes of an error in terms of the state of the transaction. The transaction could be open and committable, open and uncommittable (doomed), and no open transaction. If you want your error handling code to react differently depending on the transaction state, it's not enough to check the @@TRANCOUNT value because it only tells you whether a transaction is open or not; it doesn't tell you whether it's open and committable or doomed. T-SQL supports an alternative function called XACT_STATE that does make the distinction between the three states. It returns 0 when no transaction is open (equivalent to @@TRANCOUNT being 0), 1 when the transaction is open and committable, and -1 (minus one) when the transaction is doomed.

The following example demonstrates the outcome of a nondooming error when using TRY-CATCH:

```
SET XACT_ABORT OFF;

BEGIN TRY

  BEGIN TRAN;

  DECLARE @i AS INT = 10/0;
  -- normally there would be work here that warrants a transaction
```

```
  COMMIT TRAN;

END TRY
BEGIN CATCH

  PRINT
    CASE XACT_STATE()
      WHEN  0 THEN 'No open transaction.'
      WHEN  1 THEN 'Transaction is open and committable.'
      WHEN -1 THEN 'Transaction is doomed.'
    END;

  IF @@TRANCOUNT > 0
    ROLLBACK TRAN;

END CATCH;
```

This code generates the following output:

```
Transaction is open and committable.
```

The following example demonstrates the outcome of a dooming error when using TRY-CATCH:

```
SET XACT_ABORT OFF;

BEGIN TRY

  BEGIN TRAN;

  DECLARE @i AS INT = CAST('1,759' AS INT);
  -- normally there would be work here that warrants a transaction

  COMMIT TRAN;

END TRY
BEGIN CATCH

  PRINT
    CASE XACT_STATE()
      WHEN  0 THEN 'No open transaction.'
      WHEN  1 THEN 'Transaction is open and committable.'
      WHEN -1 THEN 'Transaction is doomed.'
    END;

  IF @@TRANCOUNT > 0
    ROLLBACK TRAN;

END CATCH;
```

This code generates the following output:

```
Transaction is doomed.
```

If you turn on the XACT_ABORT option, as recommended, most errors are treated as dooming errors. For instance, the example before the last with the divide by zero error results in a doomed transaction with XACT_ABORT turned on.

Remember, though, that in case an error with severity 20 and up happens, the connection is terminated, so any error handling code that you have doesn't have a chance to run.

MORE INFO ON CLASSIFICATION OF ERRORS AND THEIR OUTCOMES

Erland Sommarskog has a nice summary of the different classification of errors and their possible outcomes at *http://www.sommarskog.se/error_handling/Part2.html#classification*.

If you do not wish to deal differently with open and committable and doomed transactions, rather react the same in both cases, to check whether you're in an open transaction you could either check that @@TRANCOUNT is greater than zero or that XACT_STATE is different than zero. Just make sure that whatever you do, you don't leave a transaction open other than in very special, controlled, circumstances.

For example, suppose that you wanted to add error handling code to the AddOrder procedure, which you created in an earlier section. In the CATCH block you simply want to make sure that in case there's an open transaction you roll it back, and re-throw the error to the caller. Use the following code to achieve this:

```
CREATE OR ALTER PROC dbo.AddOrder
  @custid         AS INT,
  @empid          AS INT,
  @orderdate      AS DATE,
  @requireddate   AS DATE,
  @shippeddate    AS DATE,
  @shipperid      AS INT,
  @freight        AS MONEY,
  @shipname       AS NVARCHAR(40),
  @shipaddress    AS NVARCHAR(60),
  @shipcity       AS NVARCHAR(15),
  @shipregion     AS NVARCHAR(15),
  @shippostalcode AS NVARCHAR(10),
  @shipcountry    AS NVARCHAR(15),
  @OrderLines     AS dbo.OrderLines READONLY,
  @neworderid     AS INT OUT
AS

SET XACT_ABORT, NOCOUNT ON;

BEGIN TRY

BEGIN TRAN;

  -- add order header
  INSERT INTO Sales.Orders
      (custid, empid, orderdate, requireddate, shippeddate,
       shipperid, freight, shipname, shipaddress, shipcity,
       shippostalcode, shipcountry)
```

```
    VALUES
      (@custid, @empid, @orderdate, @requireddate, @shippeddate,
       @shipperid, @freight, @shipname, @shipaddress, @shipcity,
       @shippostalcode, @shipcountry);

  SET @neworderid = SCOPE_IDENTITY();

  -- add order lines
  INSERT INTO Sales.OrderDetails(orderid, productid, unitprice, qty, discount)
    SELECT @neworderid, productid, unitprice, qty, discount
    FROM @OrderLines;

  COMMIT TRAN;

END TRY
BEGIN CATCH

  IF @@TRANCOUNT > 0
    ROLLBACK TRAN;

  THROW;

END CATCH;
GO
```

Note that because you turned on the XACT_ABORT option, even if you didn't explicitly roll the transaction back in the CATCH block, the THROW command would. However, it's a good practice to keep this check in place in case someone changes the procedure in the future and removes the code that sets XACT_ABORT to on.

If you need to write any error information to a log table, make sure that you do so after the ROLLBACK TRAN statement and before the THROW command. If you try to do so before rolling the transaction back, you get an error saying that writes are not allowed under a doomed transaction. Furthermore, even if it was allowed, the ROLLBACK TRAN statement would have rolled back any such changes. Also, if you need to log data that was created by the transaction, make sure you first write it to a table variable before issuing the ROLLBACK TRAN statement, and then copy it from the table variable to the log table after rolling the transaction back. Remember that writes to a table variable are not undone when a transaction rolls back. Your catch block would look like this:

```
BEGIN CATCH
  ... declare a table variable and write what's needed into it ...
  IF @@TRANCOUNT > 0
    ROLLBACK TRAN;
  ... write what's needed to a log table, including copying data from the table variable
...
  THROW;
END CATCH;
```

When you're done, run the following code for cleanup:

```
DELETE FROM Sales.OrderDetails WHERE orderid > 11077;
DELETE FROM Sales.Orders WHERE orderid > 11077;
```

```
DBCC CHECKIDENT('Sales.Orders', RESEED, 11077);
DROP PROC IF EXISTS dbo.AddRowToT1, dbo.OuterProc, dbo.InnerProc,
  dbo.PrintErrorInfo, dbo.Divide, dbo.AddOrder, dbo.Proc1;
DROP TABLE IF EXISTS dbo.T1, dbo.ErrorLog;
```

Before continuing to the next section, remember my earlier recommendation to make sure that you read at least the first two parts in Erland Sommarskog's coverage of error handling, which you can find in his website at *http://sommarskog.se/error_handling/Part1.html*.

Skill 3.3: Implement data types and NULLs

This skill covers data types, data type conversions, and handling NULLs using the ISNULL and COALESCE functions. If some of these topics seem familiar, that's because I found it important to already cover them at least to some extent earlier in the book. For instance, I already covered some aspects of data type conversions, including implicit ones, as well as handling of NULLs, and the COALESCE and ISNULL functions in Chapter 1. Consider the coverage here as an opportunity to review these important topics again and have another chance to practice them.

This section covers how to:

- Evaluate results of data type conversions
- Determine proper data types for given data elements or table columns
- Identify locations of implicit data type conversions in queries
- Determine the correct results of joins and functions in the presence of NULL values
- Identify proper usage of ISNULL and COALESCE functions

Working with data types

When defining columns in tables, parameters in procedures and functions, and variables in T-SQL batches, you need to choose a data type for those. The data type constrains the data that is supported, in addition to encapsulating behavior that operates on the data, exposing it through operators and other means. Because data types are such a fundamental component of your data—everything is built on top—your choices of data types have dramatic implications for your application at many different layers. Therefore, this is an area that should not be taken lightly, but instead treated with a lot of care and attention.

The upcoming sections provide recommendations for choosing the appropriate data type for different database elements, choosing a data type for keys, and data type conversions.

Choosing the appropriate data type

Choosing the appropriate data types for your attributes is probably one of the most important decisions that you make regarding your data. SQL Server supports many data types from different categories: exact numeric (INT, NUMERIC), character strings (CHAR, VARCHAR), Unicode character strings (NCHAR, NVARCHAR), approximate numeric (FLOAT, REAL), binary strings (BINARY, VARBINARY), date and time (DATE, TIME, DATETIME2, SMALLDATETIME, DATETIME, DATETIMEOFFSET), and others. There are many options, so it might seem like a difficult task, but as long as you follow certain principles, you can be smart about your choices, which results in a robust, consistent, and efficient database.

One of the great strengths of the relational model is the importance it gives to enforcement of data integrity as part of the model itself, at multiple levels. One important aspect in choosing the appropriate type for your data is to remember that a type is a constraint. This means that it has a certain domain of supported values and does not allow values outside that domain. For example, the DATE type allows only valid dates. An attempt to enter something that isn't a date, like 'abc' or '20170230', is rejected. If you have an attribute that is supposed to represent a date, such as birthdate, and you use a type such as INT or CHAR, you don't benefit from built-in validating of dates. An INT type won't prevent a value such as 99999999 and a CHAR type won't prevent a value such as '20170230'.

Much like a type is a constraint, NOT NULL is a constraint as well. If an attribute isn't supposed to allow NULLs, it's important to enforce a NOT NULL constraint as part of its definition. Otherwise, NULLs find their way into your attribute.

Also, you want to make sure that you do not confuse the formatting of a value with its type. Sometimes, people use character strings to store dates because they think of storing a date in a certain format. The formatting of a value is supposed to be the responsibility of the application when data is presented. The type is a property of the value stored in the database, and the internal storage format shouldn't be your concern. This aspect has to do with the *physical data independence* principle in the relational model.

A data type encapsulates behavior. By using an inappropriate type, you miss all the behavior that is encapsulated in the type in the form of operators and functions that support it. As a simple example, for types representing numbers, the plus (+) operator represents addition, but for character strings, the same operator represents concatenation. If you chose an inappropriate type for your value, you sometimes have to convert the type (explicitly or implicitly), and sometimes juggle the value quite a bit, in order to treat it as what it is supposed to be.

Another important principle in choosing the appropriate type for your data is size. Often one of the major aspects affecting query performance is the amount of I/O involved. A query that reads less simply tends to run faster. The bigger the type that you use, the more storage it uses. Tables with many millions of rows, if not billions, are commonplace nowadays. When you start multiplying the size of a type by the number of rows in the table, the numbers can quickly become significant. As an example, suppose you have an attribute representing test scores, which are integers in the range 0 to 100. Using an INT data type for this purpose is overkill. It would use 4 bytes per value, whereas a TINYINT would use only 1 byte, and is

therefore the more appropriate type in this case. Similarly, for data that is supposed to represent dates, many people have a tendency to use the legacy DATETIME type, which uses 8 bytes of storage.

If the value is supposed to represent a date without a time, you should use DATE, which uses only 3 bytes of storage. Moreover, if the value is supposed to represent both date and time, you should consider DATETIME2. It requires storage between 6 to 8 bytes (depending on precision), and as an added value, provides a wider range of dates and improved, controllable precision. In short, you should use the smallest type that serves your needs. Though of course, this applies not in the short run, but in the long run. For example, using an INT type for a key in a table that at one point or another grows to a degree of billions of rows is a bad idea. You should be using BIGINT. But using INT for an attribute representing test scores or DATETIME for dates are both bad choices even when thinking about the long run.

Be very careful with the imprecise types FLOAT and REAL. The first two sentences in the documentation describing these types should give you a good sense of their nature: "Approximate-number data types for use with floating point numeric data. Floating point data is approximate; therefore, not all values in the data type range can be represented exactly." The benefit in these types is that they can represent very large and very small numbers beyond what any other numeric type that SQL Server supports can represent. So, for example, if you need to represent very large or very small numbers for scientific purposes and don't need complete accuracy, you may find these types useful. They're also quite economic (4 bytes for REAL and 8 bytes for FLOAT). But do not use them for things that are supposed to be precise.

I had a customer case who initially tried to use FLOAT to represent barcode numbers of products, and was then surprised by not getting the right product when scanning the products' barcodes.

To demonstrate the trickiness of these types, consider the following query, which converts a FLOAT value to NUMERIC:

```
DECLARE @f AS FLOAT = 29545428.022495;
SELECT CAST(@f AS NUMERIC(28, 14)) AS numericvalue;
```

Can you guess what the output of this code is? Here's what I got on my system:

```
numericvalue
--------------------------------------
29545428.02249500155449
```

As mentioned, some values cannot be represented precisely.

In short, make sure you use exact numeric types when you need to represent values precisely, and reserve the use of the approximate numeric types only to cases where you're certain that it's acceptable for the application.

Another important aspect in choosing a type has to do with choosing fixed types (CHAR, NCHAR, BINARY) vs. dynamic ones (VARCHAR, NVARCHAR, VARBINARY). Fixed types use the storage for the indicated size; for example, CHAR(30) uses storage for 30 characters, whether you actually specify 30 characters or less. This means that updates don't require the row to

physically expand, and therefore no data shifting is required. So for attributes that get updated frequently, where the update performance is a priority, you should consider fixed types. Note that when compression is used—specifically row compression—SQL Server stores fixed types like variable ones, but with less overhead.

Variable types use the storage for what you enter, plus a couple of bytes for offset information (or 4 bits with row compression). So for widely varying sizes of strings, if you use variable types you can save a lot of storage. As already mentioned, the less storage used, the less there is for a query to read, and the faster the query can perform. So variable length types are usually preferable in such cases when read performance is a priority.

With character strings, there's also the question of using regular character types (CHAR, VARCHAR) vs. Unicode types (NCHAR, NVARCHAR). The former use 1 byte of storage per character and support only one language (based on collation properties) besides English. The latter use 2 bytes of storage per character (unless compressed) and support multiple languages. If a surrogate pair is needed, a character requires 4 bytes of storage. So if data is in multiple languages and you need to represent only one language besides English in your data, you can benefit from using regular character types, with lower storage requirements. When data is international, or your application natively works with Unicode data, you should use Unicode data types so you don't lose information. The greater storage requirements of Unicode data are mitigated with Unicode compression.

When using types that can have a length associated with them, such as CHAR and VARCHAR, T-SQL supports omitting the length and then uses a default length. However, in different contexts, the defaults can be different. It is considered a best practice to always be explicit about the length, as in CHAR(1) or VARCHAR(30).

Choosing a data type for keys

When defining a key based on an existing attribute that already serves some purpose beyond being a key, there's usually no question about which data type to use because you've already made that choice previously. But when you need to create a new attribute solely for the purpose of being used as a key, you need to determine an appropriate data type, as well as a mechanism to generate the key values. The reality is that you hear many different opinions as to what is the best solution. Different systems and different workloads could end up with different optimal solutions. What's more, in some systems, write performance might be the priority, whereas in others, the read performance is. One solution can make the inserts faster but the reads slower, and another solution might work the other way around. At the end of the day, to make smart choices, it's important to learn the theory, learn about others' experiences, but eventually make sure that you run benchmarks in the target system.

The typical options people use to generate keys are:

- **The identity column property** A property that automatically generates keys in an attribute of a numeric type with a scale of 0; namely, any integer type (TINYINT, SMALLINT, INT, BIGINT) or NUMERIC/DECIMAL with a scale of 0.
- **The sequence object** An independent object in the database from which you can obtain new sequence values. Like identity, it supports any numeric type with a scale

of 0. Unlike identity, it's not tied to a particular column; instead, as mentioned, it is an independent object in the database. You can also request a new value from a sequence object before using it.

- **Nonsequential GUIDs** You can generate nonsequential global unique identifiers to be stored in an attribute of a UNIQUEIDENTIFIER type, which uses 16 bytes of storage. You can use the T-SQL function NEWID to generate a new GUID, possibly invoking it with a default expression attached to the column. You can also generate one from anywhere—for example, the client—by using an application programming interface (API) that generates a new GUID. The main advantage of GUIDs is that they are guaranteed to be unique across space and time. Their disadvantages are that they use a lot of storage and are quite awkward to work with.
- **Sequential GUIDs** You can generate sequential GUIDs within the machine by using the T-SQL function NEWSEQUENTIALID. This function is only allowed in a default constraint that is associated with a column.
- **Custom solutions** If you do not want to use the built-in tools that SQL Server provides to generate keys, you need to develop your own custom solution. The data type for the key then depends on your solution. An example would be storing the last used value in a table, and every time you need a new value, incrementing the existing value in the table with an UPDATE statement and using the new value when you insert a row into the target table.

One thing to consider regarding your choice of a key generator and the data type involved is the size of the data type. The bigger the type, the more storage is required, and hence the slower the reads are. A solution using an INT data type requires 4 bytes per value, BIGINT requires 8 bytes, UNIQUEIDENTIFIER requires 16 bytes, and so on. The storage requirements for your key can have a cascading effect if your clustered index is defined on the same key columns (the default for a primary key constraint). The clustered index key columns are used by all nonclustered indexes internally as the means to locate rows in the table. So if you define a clustered index on a column x, and nonclustered indexes—one on column a, one on b, and one on c—your nonclustered indexes are internally created on columns (a, x), (b, x), and (c, x), respectively. In other words, the effect is multiplied.

Regarding the use of sequential keys (as with identity, sequence, and NEWSEQUENTIALID) vs. nonsequential ones (as with NEWID or a custom randomized key generator), there are several aspects to consider.

Starting with sequential keys, all rows go into the right end of the index. When a page is full, SQL Server allocates a new page and fills it. This results in less fragmentation in the index, which is beneficial for read performance. Also, insertions can be faster when a single session is loading the data, and the data resides on a single drive or a small number of drives. However, with high-end storage subsystems, the situation can be different. When loading data from multiple sessions, such as in typical OLTP workloads like order entry systems, you end up with a performance problem known as the rightmost page latch contention. Latches are objects used to synchronize access to database pages. In this scenario, you have a bottleneck when

multiple threads try to obtain a latch against the rightmost index page and end up being queued since only one thread at a time can obtain the latch. This bottleneck prevents use of the full throughput of the storage subsystem and in systems with a high volume of small insert transactions often results in serious performance problems. As an interesting aside, the rightmost page latch contention problem exists when using the traditional disk-based architecture for data. The problem is completely eliminated when using the In-Memory OLTP feature because it uses an architecture with no locking or latching.

Consider nonsequential keys, such as random ones generated with NEWID or with a custom solution. When trying to force a row into an already full page, SQL Server performs a classic page split—it allocates a new page and moves half the rows from the original page to the new one. A page split has a cost, plus it results in index fragmentation. Index fragmentation can have a negative impact on the performance of reads. However, when using a high-end storage subsystem and loading data from multiple sessions, the random order can result in much better performance than sequential despite the splits. That's because there's no hot spot at the right end of the index, and you use the storage subsystem's available throughput better. Page splits and index fragmentation can be mitigated by periodic index rebuilds as part of the usual maintenance activities.

If for aforementioned reasons you decide to rely on keys generated in random order, you still need to decide between GUIDs and a custom random key generator solution. As already mentioned, GUIDs are stored in a UNIQUEIDENTIFIER type that is 16 bytes in size; that's large. But one of the main benefits of GUIDs is the fact that they can be generated anywhere and not conflict across time and space. You can generate GUIDs not just in SQL Server using the NEWID function, but anywhere, using APIs. Otherwise, you could come up with a custom solution that generates smaller keys that are generated in a random-like order as far as the insertions are concerned. The solution can even be a mix of a built-in tool and some tweaking on top. For example, you can find a creative solution by Wolfgang 'Rick' Kutschera at *http://dangerousdba.blogspot.com/2011/10/day-sequences-saved-world.html*. Rick uses the SQL Server sequence object, but flips the bits of the values so that the insertion is distributed across the index leaf.

To conclude this section about keys and types for keys, remember that there are multiple options. Smaller is generally better, but then there's the question of the hardware that you use, and where your performance priorities are. Also remember that although it is very important to make educated guesses, it is also important to benchmark solutions in the target environment.

Data type conversions

This section covers both explicit and implicit data type conversions and is mainly provided as a review of previously discussed topics.

You want to make sure that when indicating a literal of a type, you use the correct form. For example, literals of regular character strings are delimited with single quotation marks, as in 'abc', whereas literals of Unicode character strings are delimited with a capital N, and then single quotation marks, as in N'abc'. When an expression involves elements with different types, SQL

Server needs to apply implicit conversion when possible, and this may result in performance penalties. Note that in some cases the interpretation of a literal may not be what you think intuitively. In order to force a literal to be of a certain type, you may need to apply explicit conversion with functions like CAST, CONVERT, PARSE, or TRY_CAST, TRY_CONVERT, and TRY_PARSE. As an example, the literal 1 is considered an INT by SQL Server in any context. If you need the literal 1 to be considered, for example, a BIT, you need to convert the literal's type explicitly, as in CAST(1 AS BIT). Similarly, the literal 4000000000 is considered NUMERIC and not BIGINT. If you need the literal to be the latter, use CAST(4000000000 AS BIGINT). The difference between the functions without the TRY and their counterparts with the TRY is that those without the TRY fail if the value isn't convertible, whereas those with the TRY return a NULL in such a case. For example, the following code fails.

```
SELECT CAST('abc' AS INT);
```

It generates the following output:

```
-----------
Msg 245, Level 16, State 1, Line 28
Conversion failed when converting the varchar value 'abc' to data type int.
```

Conversely, the following code returns a NULL.

```
SELECT TRY_CAST('abc' AS INT);
```

As for the difference between CAST, CONVERT, and PARSE, with CAST, you indicate the expression and the target type; with CONVERT, there's a third argument representing the style for the conversion, which is supported for some conversions, like between character strings and date and time values. For example, CONVERT(DATE, '1/2/2017', 101) converts the literal character string to DATE using style 101 representing the United States standard. With PARSE, you can indicate the culture by using any culture supported by the Microsoft .NET Framework. For example, PARSE('1/2/2017' AS DATE USING 'en-US') parses the input literal as a DATE by using a United States English culture.

EXAM TIP

As mentioned in Chapter 1, the PARSE function is significantly slower than the CONVERT and CAST functions. So, in real life, I recommend staying away from it unless at some point Microsoft fixes the performance issue. However, when taking the exam make sure that you understand carefully what the questions are about. If presented with a conversion task and asked which solutions are correct (as opposed to efficient), by all means, you should consider using PARSE if presented with the option.

When using expressions that involve operands of different types, SQL Server usually converts the one that has the lower data type precedence to the one with the higher. Consider the expression 1 + '1' as an example. One operand is INT and the other is VARCHAR. If you look in Books Online under "Data Type Precedence (Transact-SQL)," at *https://msdn.microsoft.com/en-us/library/ms190309.aspx*, you find that INT precedes VARCHAR; hence, SQL Server

implicitly converts the VARCHAR value '1' to the INT value 1, and the result of the expression is therefore 2 and not the string '11'. Of course, you can always take control by using explicit conversion.

If all operands of the expression are of the same type, that's also going to be the type of the result, and you might not want it to be the case. For example, the result of the expression 5 / 2 in T-SQL is the INT value 2 and not the NUMERIC value 2.5, because both operands are integers, and therefore the result is an integer. If you were dealing with two integer columns, like col1 / col2, and wanted the division to be NUMERIC, you would need to convert the columns explicitly, as in CAST(col1 AS NUMERIC(12, 2)) / CAST(col2 AS NUMERIC(12, 2)).

Curiously, T-SQL handles conversions from NUMERIC to INT differently than between NUMERIC with a higher scale to a lower one. With the former, T-SQL truncates the value, with the latter, it rounds it. The following example demonstrates this:

```
SELECT
  CAST(10.999 AS NUMERIC(12, 0)) AS numeric_to_numeric,
  CAST(10.999 AS INT) AS numeric_to_int;
```

This code generates the following output:

```
numeric_to_numeric                    numeric_to_int
------------------------------------- --------------
11                                    10
```

When converting a character string to a date and time type or a date and time type with a higher precision to one with a lower precision, you get rounding—not truncation. The following example demonstrates this:

```
DECLARE
  @s AS CHAR(21) = '20170212 23:59:59.999',
  @dt2 AS DATETIME2 = '20170212 23:59:59.999999';

SELECT
  CAST(@s AS DATETIME) AS char_to_datetime,
  CAST(@dt2 AS DATETIME) AS char_to_datetime;
```

This code generates the following output:

```
char_to_datetime        char_to_datetime
----------------------- -----------------------
2017-02-13 00:00:00.000 2017-02-13 00:00:00.000
```

The above conversion behavior applies whether the conversion is explicit or implicit.

When defining attributes that represent the same thing across different tables—especially ones that are later used as join columns (like the primary key in one table and the foreign key in another)—it's very important to be consistent with the types. Otherwise, when comparing one attribute with another, SQL Server has to apply implicit conversion of one attribute's type to the other, and this could have negative performance implications, like preventing efficient use of indexes.

Handling NULLs

Chapter 1 already covered in detail the trickiness and complexity of dealing with NULLs in different T-SQL elements and provided recommendations for proper handling of NULLs. This section gives you an opportunity to practice what you know by going over a number of examples. The two main areas that this section demonstrates are using the ISNULL and COALESCE functions, and handling NULLs when combining data from different tables.

The ISNULL and COALESCE functions

The ISNULL and COALESCE functions are both commonly used functions that return the first value that is not NULL among their inputs. There are quite a few differences between them that you should be aware of before you decide which of the two is more appropriate for you to use.

The first important difference between ISNULL and COLAESCE is that the former supports only two input parameters, whereas the latter supports more than two. Here's a simple example demonstrating their use:

```
SET NOCOUNT ON;
USE TSQLV4;

DECLARE
  @x AS INT = NULL,
  @y AS INT = 1759,
  @z AS INT = 42;

SELECT COALESCE(@x, @y, @z);
SELECT ISNULL(@x, @y);
```

Both expressions return the value 1759 because in both cases it's the first value that is not NULL.

The ISNULL function is a proprietary T-SQL feature whereas the COALESCE function is defined by the ISO/ANSI SQL standard. So if there is a policy in your organization to use standard code whenever possible, you should prefer COALESCE in such a case.

The previous section in this skill discussed data type conversions, including implicit conversions. There's a curious difference between ISNULL and COALESCE in terms of how the data type of the returned value is determined and how implicit conversion is handled. With ISNULL, the data type of the result is determined like so:

1. If the first input has a data type (as opposed to being an untyped NULL literal), the result type is the type of the first input.
2. If the first input is an untyped NULL literal, and the second input has a data type, the result type is the type of the second input.
3. If both inputs are untyped NULL literals, the result data type is INT.

With COALESCE, the type is determined like so.

1. If at least one of the inputs has a type, the result type is the type with the highest precedence among the inputs.
2. If all inputs are untyped NULL literals, you get an error.

Consider the following example:

```
DECLARE
  @x AS VARCHAR(3) = NULL,
  @y AS VARCHAR(10) = '1234567890';

SELECT ISNULL(@x, @y) AS ISNULLxy, COALESCE(@x, @y) AS COALESCExy;
```

Before you actually run it, can you guess what each of the functions return?

This code generates the following output:

```
ISNULLxy COALESCExy
-------- ----------
123      1234567890
```

Notice that with ISNULL the first input determines the type, which is VARCHAR(3), and since the first input is NULL and the second input isn't, the function returns the second input value after implicitly converting it to the first input's type. Consequently, the string gets truncated after the first three characters. With COALESCE, the type of the result is the type with the highest precedence among the inputs, meaning VARCHAR(10) in our example, so the returned value doesn't get truncated.

Consider another example:

```
SELECT ISNULL('1a2b', 1234) AS ISNULLstrnum;
GO
SELECT COALESCE('1a2b', 1234) AS COALESCEstrnum;
GO
```

Again, before actually running this code, can you guess the outcome of both function calls?

With the ISNULL function, the type of the result is based on the first input, so you get the first input value with its original type as the result: '1a2b'. With the COALESCE function, the type with the highest precedence among the inputs is INT, therefore the function tries to implicitly convert the first input value to INT before returning it and fails since the value doesn't covert. This code generates the following output:

```
ISNULLstrnum
------------
1a2b

COALESCEstrnum
--------------
Msg 245, Level 16, State 1, Line 85
Conversion failed when converting the varchar value '1a2b' to data type int.
```

Another difference between the two functions has to do with the nullability of the result column when using them in a SELECT INTO statement. With ISNULL, if any of the input expressions is nonnullable, the result column is defined as NOT NULL. If both inputs are nullable, the result is a column defined as allowing NULLs. With COALESCE, only if all inputs are nonnullable, the result column is defined as NOT NULL, otherwise, it is defined as allowing NULLs.

The following code demonstrates the difference between the two functions in terms of the nullability of the result:

```
DROP TABLE IF EXISTS dbo.TestNULLs;
GO
SELECT empid,
  ISNULL(region, country) AS ISNULLregioncountry,
  COALESCE(region, country) AS COALESCEregioncountry
INTO dbo.TestNULLs
FROM HR.Employees;

SELECT
  COLUMNPROPERTY(OBJECT_ID('dbo.TestNULLs'), 'ISNULLregioncountry',
    'AllowsNull') AS ISNULLregioncountry,
  COLUMNPROPERTY(OBJECT_ID('dbo.TestNULLs'), 'COALESCEregioncountry',
    'AllowsNull') AS COALESCEregioncountry;

DROP TABLE IF EXISTS dbo.TestNULLs;
```

The region column in the HR.Employees table allows NULLs and the country column doesn't. With ISNULL, because at least one input is defined as not allowing NULLs, the result column is defined as NOT NULL. With COALESCE, because at least one input allows NULLs, the result column allows NULLs as well. Here's the output of this code indicating the nullability of the two result columns:

```
ISNULLregioncountry COALESCEregioncountry
------------------- ---------------------
0                   1
```

Another important difference between the two functions has to do with performance when using subqueries. That is, when comparing an expression such as ISNULL((<subquery>), 0) with COALESCE((<subquery>), 0). The ISNULL function evaluates the subquery only once. If its result is not NULL, it returns its result. If it is NULL, it evaluates the second input and returns its result. With COALESCE, according to the SQL standard, the expression is translated to:

```
CASE WHEN (<subquery>) IS NOT NULL THEN (<subquery>) ELSE 0 END
```

If the result of the execution of the subquery in the WHEN clause isn't NULL, SQL Server executes it a second time in the THEN clause. In other words, in such a case it executes it twice. Only if the result of the execution in the WHEN clause is NULL, SQL Server doesn't execute the subquery again, rather returns the ELSE expression. So when using subqueries, the ISNULL function has a performance advantage.

EXAM TIP

If your exam includes questions involving ISNULL and COALESCE, chances are that you are expected to know how to pick between the two based on the differences between them. For your convenience, Table 3-2 provides a comparison summary between the two functions. Make sure you understand and memorize all items in the table.

TABLE 3-2 Comparison between ISNULL and COALESCE

aspect	isnull	coalesce
Number of supported parameters	2	> 2
Standard	No	Yes
Data type of result	1. If first input has a type, that's the type of the result. 2. Otherwise, if second input has a type, that's the type of the result. 3. If both inputs are untyped NULL literals, the result type is INT.	1. If at least one input has a type, the result type is the type with the highest precedence. 2. If all inputs are untyped NULL literals, you get an error.
nullability of result	If any input is nonnullable, result is defined as NOT NULL, otherwise as NULL.	If all inputs are nonnullable, result is defined as NOT NULL, otherwise as NULL.
Might execute subquery more than once	No	Yes

Handling NULLs when combining data from multiple tables

When you need to combine data from multiple tables you use tools like joins, subqueries, and set operators. You always want to check if NULLs are possible in the tables you're combining, especially when it concerns the columns that you're comparing between the sides. If NULLs are possible in the data, you want to make sure that you have the right logic in place in your code to get the behavior that you consider as correct. This section demonstrates NULL handling when combining data from multiple tables with all three tools.

The examples in this section query tables called TableA and TableB. Run the following code to create these tables and populate them with sample data:

```
DROP TABLE IF EXISTS dbo.TableA, dbo.TableB;
GO
CREATE TABLE dbo.TableA
(
  key1 CHAR(1) NOT NULL,
  key2 CHAR(1) NULL,
  A_val VARCHAR(10) NOT NULL,
  CONSTRAINT UNQ_TableA_key1_key2 UNIQUE CLUSTERED (key1, key2)
);

INSERT INTO dbo.TableA(key1, key2, A_val)
  VALUES('w', 'w', 'A w w'),
        ('x', 'y', 'A x y'),
        ('x', NULL, 'A x NULL');
```

```
CREATE TABLE dbo.TableB
(
  key1 CHAR(1) NOT NULL,
  key2 CHAR(1) NULL,
  B_val VARCHAR(10) NOT NULL,
  CONSTRAINT UNQ_TableB_key1_key2 UNIQUE CLUSTERED (key1, key2)
);

INSERT INTO dbo.TableB(key1, key2, B_val)
  VALUES('x', 'y', 'B x y'),
        ('x', NULL, 'B x NULL'),
        ('z', 'z', 'B z z');
```

Joins and subqueries behave similarly in terms of NULLs when you compare key columns between the two sides of the operation in the join predicate or the subquery's WHERE clause. When comparing two columns using an equality operator (=), if either side is NULL, the outcome of the comparison is the logical value *unknown*. As part of a join predicate, the case is considered a nonmatch, and as part of a filter, the result row is discarded. If you want to get a match/keep the row when both sides are NULL, you need to add explicit special handling.

I'll demonstrate this behavior first with joins and then with subqueries. The following example joins TableA and TableB and compares both key1 (nonnullable) and key2 (nullable) from both sides:

```
SELECT A.A_val, B.B_val
FROM dbo.TableA AS A
  INNER JOIN dbo.TableB AS B
    ON A.key1 = B.key1
    AND A.key2 = B.key2;
```

Only rows with proper key values that match are returned. That is, only rows where both keys are not NULL and equal are considered matches and therefore are returned. This code generates the following output:

```
A_val      B_val
---------- ----------
A x y      B x y
```

Notice that even though both sides have a row where key1 is x and key2 is NULL, such a case is not considered a match, and therefore a result row is not returned for it. If you want such a case to be considered a match, you need to add special handling of the NULLs. One way to achieve this is to add a predicate that checks for the option that both sides are NULL, like so:

```
SELECT A.A_val, B.B_val
FROM dbo.TableA AS A
  INNER JOIN dbo.TableB AS B
    ON A.key1 = B.key1
    AND (A.key2 = B.key2 OR A.key2 IS NULL AND B.key2 IS NULL);
```

This special handling is only required for the nullable columns; therefore, you only need it for key2, but not for key1. This code generates the following output:

```
A_val      B_val
---------- ----------
A x NULL   B x NULL
A x y      B x y
```

This form is considered order-preserving by the optimizer, meaning that it enables efficient use of indexing.

Another common solution is to use the ISNULL or COALESCE functions to replace a NULL with a value that cannot normally appear in the data in both sides. This way, when both sides are NULL, you get a match. Here's how you use this technique in our example:

```
SELECT A.A_val, B.B_val
FROM dbo.TableA AS A
  INNER JOIN dbo.TableB AS B
    ON A.key1 = B.key1
    AND COALESCE(A.key2, '<N/A>') = COALESCE(B.key2, '<N/A>');
```

This code generates the following output:

```
A_val      B_val
---------- ----------
A x NULL   B x NULL
A x y      B x y
```

Unfortunately, this form is not considered order-preserving by the optimizer, and therefore might result in a less optimal plan compared to the previous solution. But as mentioned several times in this guide, during the exam make sure you understand carefully what the exact requirements of the questions are. When only asked about the correctness of a solution and not its efficiency, use of such techniques is fair game. When considering which solution to use in production code, you want to stay away from such techniques.

As mentioned, subqueries behave very similar to joins in terms of NULL handling. Consider the following example:

```
SELECT A.A_val
FROM dbo.TableA AS A
WHERE EXISTS
  ( SELECT * FROM dbo.TableB AS B
    WHERE A.key1 = B.key1
      AND A.key2 = B.key2 );
```

Like with the first example with the join, without any special NULL handling, only rows from TableA with proper key values (ones that are not NULL) that match all compared columns in TableB are filtered. The row in Table A where key1 is x and key2 is NULL is not returned, despite the fact that Table B also has a row where key1 is x and key2 is NULL. This code generates the following output:

```
A_val
----------
A x y
```

If you are supposed to return such rows, just like with joins, you need to add special explicit NULL treatment. Here's an example demonstrating the order-preserving technique with the extra predicate:

```
SELECT A.A_val
FROM dbo.TableA AS A
WHERE EXISTS
  ( SELECT * FROM dbo.TableB AS B
    WHERE A.key1 = B.key1
      AND (A.key2 = B.key2 OR A.key2 IS NULL AND B.key2 IS NULL) );
```

This code generates the following output:

```
A_val
----------
A x NULL
A x y
```

And here's an example demonstrating the technique that is not order-preserving, and therefore typically less efficient, with the COALESCE function:

```
SELECT A.A_val
FROM dbo.TableA AS A
WHERE EXISTS
  ( SELECT * FROM dbo.TableB AS B
    WHERE A.key1 = B.key1
    AND COALESCE(A.key2, '<N/A>') = COALESCE(B.key2, '<N/A>') );
```

This code generates the following output:

```
A_val
----------
A x NULL
A x y
```

Using a set operator, like INTERSECT, the comparison between the two inputs uses a concept of distinctness instead of equality. With distinctness, when you compare two NULLs you get *true* and not *unknown* as the result. That is, it is true that one NULL isn't distinct from another NULL.

Here's an example combining the rows with key1 and key2 from the two tables using the INTERSECT operator:

```
SELECT key1, key2 FROM dbo.TableA
INTERSECT
SELECT key1, key2 FROM dbo.TableB;
```

This code generates the following output:

```
key1 key2
---- ----
x    NULL
x    y
```

As you can see, there's no need here for special handling of NULLs if you want to get a true when comparing two NULLs. That's the behavior of set operators by design. The one drawback that set operators have compared to the alternative tools is that with set operators you're limited to returning only the columns that you're comparing.

In Chapter 1 I described a technique that brings together all three tools to combine data between tables in a manner that is order-preserving, allows returning elements from both sides and that uses distinctness-based semantics. Here's the technique applied to our example:

```
SELECT A.A_val, B.B_val
FROM dbo.TableA AS A
  INNER JOIN dbo.TableB AS B
    ON EXISTS( SELECT A.key1, A.key2
               INTERSECT
               SELECT B.key1, B.key2 );
```

This code generates the following output:

```
A_val      B_val
---------- ----------
A x NULL   B x NULL
A x y      B x y
```

This beautiful technique is considered pretty advanced and is not very well known, and hence it's not very likely that it will show up in the exam. However, if you do understand it well, you should have a pretty good grasp of the fundamental querying tools that it's based on. And what could be a nicer way to end the book than with this example?

When you're done, run the following code for cleanup:

```
DROP TABLE IF EXISTS dbo.TableA, dbo.TableB;
```

Chapter summary

- Views are reusable table expressions that don't support parameters. Inline table valued functions (TVFs) are like views, but with parameter support. You should think of inline TVFs as parameterized views.
- Scalar user-defined functions (UDFs) accept parameters and return a single value. You cannot change data from UDFs or have any side effects on the database. T-SQL does not support error handling in UDFs.
- Multistatement table-valued UDFs define a table variable that they return in their header. The responsibility of the UDF's body is to fill the returned table variable with data. When you query such a UDF, SQL Server internally creates the table variable, runs the function's flow to fill it with data, and hands it to the calling query. The benefit of UDFs is that unlike with stored procedures you can interact with them as part of a query.

- Stored procedures are reusable routines that allow you to encapsulate logic. Unlike UDFs, stored procedures can modify data in the database, use dynamic SQL, and support error handling. They help you hide the complexity of your tasks. They also help you better control permissions by granting EXECUTE permissions to the user on the procedure without granting permissions to perform the underlying activities directly. Unlike UDFs, stored procedures cannot appear in a query.
- SQL Server supports doing work as transactions with full ACID properties (atomicity, consistency, isolation and durability).
- T-SQL supports a TRY-CATCH construct for error handling purposes. Using this construct appropriately, combined with making sure that you don't leave a transaction open in case of errors, allows you to provide consistent error handling solutions.
- By turning the XACT_ABORT option to on you get a more consistent outcome in terms of an open transaction when an error happens. When not using TRY-CATCH, an open transaction is rolled back and the batch is aborted. When using TRY-CATCH, the transaction is doomed and control passes to the CATCH block.
- The ISNULL and COALESCE functions return the first value that is not NULL among their inputs.
- When combining data from multiple tables you use tools like joins, subqueries and set operators. If NULLs are possible in the key columns that you're matching, you need to think about the desired outcome of the comparison. If you wish to get a match when comparing two NULLs, with joins and subqueries you need to add special handling, whereas with set operators you don't.

Thought experiment

In this thought experiment, demonstrate your skills and knowledge of the topics covered in this chapter. You can find the answer to this thought experiment in the next section.

You are a member of a database development team of a large retail company. There seems to be some confusion among the developers in terms of which tools to use for which tasks, resulting in inconsistent choices. Your team has a meeting where different members raise questions with regards to the recommended tools and practices to use. You are asked about your opinion concerning the following questions:

1. Some of the developers argue against implementing logic in database routines like stored procedures. Do you have any arguments in favor of using those?
2. Some of the developers argue that there's no need for error handling in T-SQL, rather all error handling should be done in the application. Can you provide any arguments in favor of handling errors in T-SQL, and any recommendations for consistent error handling?

3. There's some inconsistency in the choice of data types for columns that represent the same thing in different tables. Making a change in the type of a column involves some refactoring, and in some cases downtime. What are the benefits and arguments in favor of creating a plan for more consistent use of data types for attributes that represent the same thing?
4. Currently there's a lot of inconsistency in using ISNULL and COALESCE. If the dev team had to pick one of the two to be consistent, which of the two should it be? Also, are there special cases where an exception should be made?
5. When joining data with keys that support NULLs, developers regularly use the ISNULL function in both sides of the join to replace a NULL with an alternative value, and this way get a match when both sides are NULL. However, users complain about bad performance for those queries. Can you suggest an alternative solution?

Thought experiment answer

This section contains the solution to the thought experiment.

1. Stored procedures allow you to encapsulate logic in a reusable database routine. Compared to implementing logic in the application, it's much easier to deploy a change in a stored procedure. You simply alter the procedure with the new code, and from that moment everyone starts using the new version. Stored procedures also tend to result in less network traffic since the application passes through the network only the procedure name and its arguments; the flow runs in the database, and the stored procedure returns only the final result through the network. Stored procedures also make it easy to handle security because you grant the user only EXECUTE permissions on the stored procedure, without needing to grant the user direct permissions for the underlying activities.
2. The two don't have to be mutually exclusive. For highly robust solutions you want to handle errors in both the application and the T-SQL code. You always want to check the inputs in the application and avoid submitting any unnecessary activity to the database. Once the request is submitted to the database, when an error happens, T-SQL doesn't behave very consistently by default in terms of the transaction and batch state. Using the TRY-CATCH construct, plus turning on the XACT_ABORT option, you can achieve a more consistent behavior. When an error happens, the transaction is doomed and control passes to the CATCH block. After doing what you need to do in the CACTH block, you typically want to roll back the transaction, if still open, and rethrow the error to the caller. You can then do what you need to do in the application with the error.

3. Columns that represent the same thing in different tables are often used as join columns, such as in a foreign key-primary key relationship. Such columns are typically used as join columns. Inconsistency in the data types between the sides of the join causes SQL Server to implicitly convert the type of one of the sides to the other. With implicit conversion, the data loses its original ordering property, preventing SQL Server from being able to rely on index ordering to support the join. Therefore, it's important to use consistent types, and if currently inconsistent, to create a plan for altering the types so that eventually they are consistent.
4. The ISNULL function is proprietary whereas COALESCE is standard. Furthermore, ISNULL supports only two inputs whereas COALESCE supports more than two. For these reasons, if you had to choose only one for consistency, COALESCE should be preferred. The one exception is when you do have only two inputs, and the first is a subquery, the ISNULL function executes the subquery only once whereas the COALESCE function might execute it twice (when the result is not NULL). So, if performance is a concern, in this exceptional case it's recommended to use ISNULL.
5. Once you apply manipulation to a column like with the ISNULL and COALESCE functions, the data loses its original ordering property—at least as far as the optimizer is concerned. This means that you prevent the optimizer from being able to rely on the ordering of supporting indexes. So instead of COALESCE(T1.key1, <somevalue>) = COALESCE(T2.key1, <somevalue>), you should prefer T1.key1 = T2.key1 OR (T1.key1 IS NULL AND T2.key1 IS NULL).

Index

Symbols

A

B

C

D

E

F

G

H

I

J

K

L

M

N

O

P

Q

R

S

T

U

V

W

X

About the author

ITZIK BEN-GAN is a T-SQL instructor for and co-founder of SolidQ. A Microsoft Data Platform MVP (Most Valuable Professional) since 1999, Itzik has delivered numerous training events around the world focused on T-SQL querying, query tuning, and programming. Itzik has authored many T-SQL books as well as a monthly column for SQL Server Pro. Itzik's speaking activities include SQLPASS, SQLBits, IT/Dev Connections, and various user groups around the world. Itzik is the author of SolidQ's Advanced T-SQL Querying, Programming and Tuning, and T-SQL Fundamentals courses, along with being a primary resource within the company for its T-SQL-related activities.