THE EXPERT'S VOICE® IN SQL SERVER

FOURTH EDITION

SQL Server T-SQL Recipes

GET THE JOB DONE WITH SQL SERVER'S POWERFUL DATABASE PROGRAMMING AND QUERY LANGUAGE

Jason Brimhall, Jonathan Gennick, Wayne Sheffield

Apress®

Contents at a Glance

About the Authors lxxiii
About the Technical Reviewer lxxv
Acknowledgments lxxvii
Introduction lxxix

■ Chapter 1: Getting Started with SELECT 1
■ Chapter 2: Elementary Programming 27
■ Chapter 3: Working with NULLS 51
■ Chapter 4: Querying from Multiple Tables 67
■ Chapter 5: Aggregations and Grouping 91
■ Chapter 6: Advanced Select Techniques 115
■ Chapter 7: Windowing Functions 141
■ Chapter 8: Inserting, Updating, Deleting 173
■ Chapter 9: Working with Strings 213
■ Chapter 10: Working with Dates and Times 233
■ Chapter 11: Working with Numbers 255
■ Chapter 12: Transactions, Locking, Blocking, and Deadlocking 279
■ Chapter 13: Managing Tables 313
■ Chapter 14: Managing Views 347
■ Chapter 15: Managing Large Tables and Databases 367
■ Chapter 16: Managing Indexes 389

Chapter 17: Stored Procedures 417
Chapter 18: User-Defined Functions and Types 437
Chapter 19: In-Memory OLTP 473
Chapter 20: Triggers 495
Chapter 21: Error Handling 531
Chapter 22: Query Performance Tuning 551
Chapter 23: Hints 599
Chapter 24: Index Tuning and Statistics 613
Chapter 25: XML 639
Chapter 26: Files, Filegroups, and Integrity 663
Chapter 27: Backup 703
Chapter 28: Recovery 733
Chapter 29: Principals and Users 761
Chapter 30: Securables, Permissions, and Auditing 799
Chapter 31: Objects and Dependencies 859
Index 873

Introduction

Sometimes all one wants is a good example.

T-SQL is fundamental to working with SQL Server. Almost everything you do, from querying a table to creating indexes to backing up and recovering, ultimately comes down to T-SQL statements being issued and executed. Sometimes a utility executes statements on your behalf; other times you must write them yourself.

And when you have to write them yourself, you're probably going to be in a hurry. Information technology is like that. It's a field full of stress and deadlines, and don't we all just want to get home for dinner with our families?

We sure do want to be home for dinner, and that brings us full circle to the example-based format you'll find in this book. If you have a job to do that's covered in this book, you can count on a clear code example and very few words to waste your time. We put the code first! And explain it afterward. We hope our examples are clear enough that you can just crib from them and get on with your day, but the detailed explanations are there if you need them.

We've missed a few dinners from working on this book. We hope it helps you avoid the same fate.

Who This Book Is For

SQL Server T-SQL Recipes is aimed at developers deploying applications against Microsoft SQL Server 2012 and 2014. The book also helps database administrators responsible for managing those databases. Any developer or administrator valuing good code examples will find something of use in this book.

Conventions

Throughout the book, we've tried to keep to a consistent style for presenting SQL and results. Where a piece of code, a SQL reserved word, or a fragment of SQL is presented in the text, it is presented in fixed-width Courier font, such as this example:

```
SELECT * FROM HumanResources.Employee;
```

Where we discuss the syntax and options of SQL commands, we use a conversational style so you can quickly reach an understanding of the command or technique. We have chosen not to duplicate complex syntax diagrams that are best left to the official, vendor-supplied documentation. Instead, we take an example-based approach that is easy to understand and adapt.

Downloading the Code

The code for the examples shown in this book is available on the Apress web site, `www.apress.com`. A link can be found on the book's information page (`www.apress.com/9781484200629`) on the Source Code/Downloads tab. This tab is located in the Related Titles section of the page.

CHAPTER 1

Getting Started with SELECT

by Jonathan Gennick

Transact-SQL is a proprietary implementation of the SQL language. It is often referred to simply as T-SQL, and you'll see us calling it by that shorter name throughout this book. The T-SQL language extends SQL by adding procedural syntax that is useful in programming both application and business logic to run inside the database server. There's much to learn, and it all begins right here with SELECT.

Tip You can find and download various editions of the Adventure Works example database from http://msftdbprodsamples.codeplex.com/.

1-1. Connecting to a Database

Problem

You are working from the command line, or maybe you just prefer to work using commands, even from the SQL Server Management Studio GUI, and you wish to connect to a specific database. For example, you wish to connect to the example database used throughout this book.

Solution

Execute the USE statement and specify the name of your target database. The following example connects to the Adventure Works example database used in this book:

```
USE AdventureWorks2014;
```

```
Command(s) completed successfully.
```

The success message indicates a successful connection. You may now execute queries against tables and views in the database without having to qualify those object names by specifying the database name each time.

How It Works

When you first launch SQL Server Management Studio you are connected to a default database that your administrator has associated with your login. By default, that default database is the so-called *master* database. Being connected to the master is usually not convenient, and you shouldn't be storing your data in that database. Executing a USE statement lets you more easily access tables and views in the database you're intending to use, and there is the added benefit of your being less likely to mistakenly create objects in the master database.

1-2. Checking the Database Server Version

Problem

You've connected to a database instance and have no idea whether that instance represents SQL Server 2014, SQL Server 2012, or something even more ancient from the Prekatmai or Precambrian eras.

Solution

Query the instance for its version information. Do that by invoking the @@VERSION function. For example:

```
SELECT @@VERSION;
```

```
-----------------------------------------------------
Microsoft SQL Server 2014 - 12.0.2000.8 (X64)
...
```

How It Works

There's no getting around it. You can see that we're running Community Technolgy Preview 2 (CTP2) while revising the book. That's because we want the book to be done shortly after the production release. We write against CTP2, and we test a second time against the Release to Manufacturing (RTM) just prior to publication. We then run every example again after the RTM is released to be sure nothing has changed.

1-3. Checking the Database Name

Problem

You want to determine via a query which database you are connected to. You can look up at the title bar when running SQL Server Management Studio, but today you happen to be running sqlcmd from the Windows command prompt. You want to be reminded of which database you specified in your most recent USE command.

Solution

Query for the name of the database currently being used. For example:

```
select DB_NAME();
```

```
------
master
```

How It Works

I surprised myself when generating the solution example in this recipe. I had thought I was using the Adventure Works database. I came in this morning and woke my PC from sleep, remembering that I had been in Adventure Works yesterday evening. Management Studio threw up a Connect to Server dialog, and I reflexively hit the Connect button while still in that first-cup-of-coffee state of mind. Mentally, I was still set in Adventure Works. But in reality, I had just connected to the master database by default. The sleep/wake cycle had broken my connection from yesterday, and I was genuinely surprised at seeing the result from this recipe's example.

So be careful!

Keep track of what database you are using as your default. Keep an eye on the title bar when executing queries from SQL Server Management Studio. Query as shown in this recipe if you're ever not sure and are executing from the command line. It's not fun to unleash a SQL statement against the wrong database, and it's especially not fun when said statement actually executes.

■ **Note** You'll learn more about `sqlcmd` in Chapter 2. It's a utility that's useful for executing T-SQL in batch mode.

1-4. Checking Your Username

Problem

You want to access your current username from SQL, either to remind yourself of who you are logged in as, or to record the name as part of a logging solution.

Solution

There are actually three names to be concerned about. There is your login name that you used when authenticating to SQL Server. There is your database username that you are associated with upon successfully logging in. Lastly, there is the username providing the credentials under which any queries are executed. Query for the names by invoking the `ORIGINAL_LOGIN()`, `CURRENT_USER`, and `SYSTEM_USER` functions respectively, as follows:

```
SELECT ORIGINAL_LOGIN(), CURRENT_USER, SYSTEM_USER;
```

```
-------------------------- -------- --------------------------

GennickT410\JonathanGennick dbo      GennickT410\JonathanGennick
```

How It Works

The example shows that I authenticated to SQL Server using the name `GennickT410\JonathanGennick`. That's my Windows login, made up from my PC name followed by my Windows username. The path-like syntax is typical of what you will see when Windows authentication is used. Otherwise, if you see just a simple name having no path-like syntax, you can be reasonably certain that SQL Server authentication was used, and that the login name and password were authenticated, not by Windows, but by the database engine.

After authenticating to SQL Server, you are then associated with a database username. This is the username that matters for object permissions. In the example, my database username is given as dbo.

Database administrators sometimes impersonate other users when testing queries. When doing that, the `SYSTEM_USER` function returns the name of the user being impersonated. However, the call to `ORIGINAL_LOGIN()` always returns the name used when first authenticating to the instance.

■ **Caution** Use of T-SQL's `EXECUTE AS` syntax to impersonate another user will cause `SYSTEM_USER` and `CURRENT_USER` to return the login name and database name of the user who is being impersonated. That is done by design, and is something to be aware of.

1-5. Querying a Table

Problem

You have a table or a view. You wish to retrieve data from specific columns.

Solution

Write a `SELECT` statement. List the columns you wish returned following the `SELECT` keyword. For example:

```
SELECT NationalIDNumber,
       LoginID,
       JobTitle
FROM HumanResources.Employee;
```

```
NationalIDNumber LoginID                   JobTitle
---------------- ------------------------- ----------------------------
295847284        adventure-works\ken0      Chief Executive Officer
245797967        adventure-works\terri0    Vice President of Engineering
509647174        adventure-works\roberto0  Engineering Manager
...
```

Specify an asterisk (*) instead of a list to return all of the columns. Here's an example showing that syntax:

```
SELECT  *
FROM  HumanResources.Employee;
```

```
BusinessEntityID NationalIDNumber LoginID                  OrganizationNode ...
---------------- ---------------- -----------------------   ---------------- ...
1                295847284        adventure-works\ken0       0x              ...
2                245797967        adventure-works\terri0     0x58            ...
3                509647174        adventure-works\roberto0   0x5AC0          ...
...
```

How It Works

The FROM clause names the table to be queried. Data is returned from that table. The comma-delimited list following the SELECT keyword specifies the columns to be returned. Whitespace doesn't matter. You can list the columns one per line as in the example, or you can list them all on the same line.

Specifying an asterisk (*) instead of a column list returns all columns of the table you are querying. Using that syntax is handy when writing ad-hoc queries and executing them from Management Studio. Don't use it from program code though. Doing so can put your program at risk of failure due to future column additions to the table, and even due to a simple rearranging of the existing columns. You're also likely to negatively impact performance by returning more data over the network than is needed. Protect yourself from both problems by listing only those columns that are really needed by the program you're writing. Don't return anything unnecessary.

1-6. Returning Specific Rows

Problem

You want to restrict query results to a subset of rows in the table that interest you.

Solution

Specify a WHERE clause that gives the conditions that rows must meet in order to be returned. For example, the following query returns only rows in which the person's title is "Ms."

```
SELECT  Title, FirstName, LastName
FROM Person.Person
WHERE Title = 'Ms.';
```

```
Title    FirstName   LastName
-------- ----------- -----------
Ms.      Gail        Erickson
Ms.      Janice      Galvin
Ms.      Jill        Williams
...
```

You may combine multiple conditions in one clause through the use of the logical operators AND and OR. The following query looks specifically for Ms. Antrim's data:

```
SELECT  Title, FirstName, LastName
FROM  Person.Person
WHERE Title = 'Ms.' AND LastName = 'Antrim';
```

```
Title    FirstName   LastName
-------- ----------- -----------
Ms.      Ramona      Antrim
```

How It Works

The WHERE clause provides search conditions that determine the rows to be returned by the query. Search conditions are written as predicates, which are expressions that evaluate to TRUE, FALSE, or UNKNOWN. Only rows for which the final evaluation of the WHERE clause is TRUE are returned. Table 1-1 lists some of the commonly used comparison operators that are available.

Table 1-1. *Operators*

Operator	Description
!=	Tests two expressions not being equal to each other.
!>	Tests that the left condition is not greater than the expression to the right.
!<	Tests that the right condition is not less than the expression to the right.
<	Tests the left condition as less than the right condition.
<=	Tests the left condition as less than or equal to the right condition.
<>	Tests two expressions not being equal to each other.
=	Tests equality between two expressions.
>	Tests the left condition being greater than the expression to the right.
>=	Tests the left condition being greater than or equal to the expression to the right.

Don't think of a WHERE clause as going out and retrieving rows that match the conditions. Think of it as a fish net or a sieve. All the possible rows are dropped into the net. Unwanted rows fall through. When a query is done executing, the rows remaining in the net are those that match the predicates you listed. Database engines will optimize execution, but the fish-net metaphor is a useful one when initially crafting a query.

You may combine multiple search conditions by utilizing the AND and OR logical operators. The AND logical operator joins two or more search conditions and returns rows only when each of the search conditions is TRUE. The OR logical operator joins two or more search conditions and returns rows when any one of the conditions are true. The second solution example shows the following AND operation. Both search conditions must be true for a row to be returned in the result set. Thus, only the row for Ms. Antrim is returned.

```
WHERE Title = 'Ms.' AND LastName = 'Antrim'
```

Use the OR operator to specify alternate choices. Use parentheses to clarify the order of operations. The following example shows an OR expression involving two LastName values. It is the result from that OR expression that is passed to the AND expression.

```
WHERE Title = 'Ms.' AND
      (LastName = 'Antrim' OR LastName = 'Galvin')
```

UNKNOWN values can make their appearance when NULL data is accessed in the search condition. A NULL value doesn't mean that the value is blank or zero—only that the value is unknown. Recipe 1-15 later in this chapter shows how to identify rows either having or not having NULL values.

1-7. Listing the Available Tables

Problem

You want to programmatically list the names of available tables in a schema. You can see the tables from Management Studio, but you want them from T-SQL as well.

Solution

One approach is to query the information schema views. This is an ISO standard approach. For example, execute the following query to see a list of all the tables and views in the HumanResources schema:

```
SELECT table_name, table_type
FROM information_schema.tables
WHERE table_schema = 'HumanResources';
```

```
TABLE_NAME                     TABLE_TYPE
------------------------------ ----------
Shift                          BASE TABLE
Department                     BASE TABLE
...
vJobCandidate                  VIEW
vJobCandidateEmployment        VIEW
```

You may also choose to forget about following ISO standard, and query the system catalog instead. The relevant views are sys.tables for tables and sys.views for views. For example

```
SELECT name
FROM sys.tables
WHERE SCHEMA_NAME(schema_id)='HumanResources';
```

```
name
-----------
Shift
Department
Employee
...
```

How It Works

The information schema views are designed to be friendly toward interactive querying. They also have the advantage of conforming to the ISO standard. There are a number of such views. The one queried in the example is `information_schema.tables`. It returns information about tables, and also about views.

There are also system catalog views. These are more detailed than the information schema views, and they can be a bit less friendly to query. For example, the `sys.tables` view doesn't return a schema name in friendly text form as `information_schema.tables` does. Instead, you get a schema ID number. That's why the second solution example had to invoke the function `SCHEMA_NAME(..)` in the `WHERE` clause, so as to translate the ID into a readable name:

```
where schema_name(schema_id)='HumanResources'
```

The information schema treats a view as a subtype of a table. The term *base table* refers to what in SQL Server is a table, and a *view* is a stored query referencing the base tables, and possibly other views. SQL Server's system catalog returns information about views and tables through separate catalog views. The first solution example returns table and view names, whereas the second returns only table names.

Why might you wish for programmatic access to metadata? One use of such access is to write SQL statements that create groups of news statements to be executed. Say, for example, that you wish to drop all the tables in the human resources schema. You can choose to create all the `DROP` statements through a query:

```
SELECT 'DROP ' + table_schema + '.' + table_name + ';'
FROM information_schema.tables
WHERE table_schema = 'HumanResources'
  AND table_type = 'BASE TABLE';
```

```
-------------------------------
DROP HumanResources.Shift;
DROP HumanResources.Department;
...
```

Then you can copy the results, paste them in as the next query batch, and hit execute, and your tables are gone.

Using PowerShell might sometimes be a better way to get the job when needing to operate on groups of objects all in one go, in a set-oriented manner. However, the quick-and-dirty technique of using SQL to write SQL can be handy too.

1-8. Naming the Output Columns

Problem

You don't like the column names returned by a query. You wish to change the names for clarity in reporting, or to be compatible with an already-written program that is consuming the results from the query.

Solution

Designate what are called *column aliases*. Use the `AS` clause for that purpose. For example:

```
SELECT BusinessEntityID AS "Employee ID",
   VacationHours AS "Vacation",
   SickLeaveHours AS "Sick Time"
FROM HumanResources.Employee;
```

```
Employee ID Vacation Sick Time
----------- -------- ---------
          1       99        69
          2        1        20
          3        2        21
```

How It Works

Each column in a result set is given a name. The name appears in the column heading when you execute a query ad-hoc using Management Studio. The name is also the name by which any program code must reference the column when consuming the results from a query. You can specify any name you like for a column via the AS clause. Such a name is termed a *column alias*.

There are some syntax alternatives to be aware of, which you might encounter when looking over existing code written by others. The following lines show these variations, and all have the same effect:

```
BusinessEntityID AS "Employee ID"
BusinessEntityID "Employee ID"
BusinessEntityID AS [Employee ID]
```

The first two lines show ISO standard syntax. The third is syntax specific to SQL Server. The first line shows the use of the AS clause, which represents the latest thinking in the standard, and thus we recommend that approach.

■ **Note** You can omit the enclosing quotes around a column alias when there are no spaces involved.

1-9. Providing Shorthand Names for Tables

Problem

You are writing a complicated WHERE clause, or a SELECT list, mixing column names from many tables, and it is becoming tedious to properly qualify each column name with its associated table and schema name.

Solution

Specify a table alias for each table in your query. Use the AS keyword to do that. For example:

```
SELECT E.BusinessEntityID AS "Employee ID",
   E.VacationHours AS "Vacation",
   E.SickLeaveHours AS "Sick Time"
FROM HumanResources.Employee AS E
WHERE E.VacationHours > 40;
```

How It Works

Specify table aliases using the AS clause. Place an AS clause immediately following each table name in the query's FROM clause. The solution example provides the alias, or alternate name, E for the table HumanResources.Employee. As far as the rest of the query is concerned, the table is now named E. By extension, you must now yourself refer to the table only as E.

Table aliases make it easy to qualify column names in a query. It is much easier to type

```
E.BusinessEntityID
```

than it is to type

```
HumanResources.Employee.BusinessEntityID
```

In real-life use, and especially in large queries, it is helpful to make your aliases more readable than the ones in our example. For example, specify Emp instead of E as the alias for the Employee table. It is easier to remember later what Emp means than to struggle over the single letter E.

Table aliases work much like column aliases, so be sure to read Recipe 1-8 as well. The syntax alternatives described in that recipe also apply when designating table aliases.

1-10. Computing New Columns from Existing Data

Problem

You are querying a table that lacks the precise bit of information you need. However, you are able to write an expression to generate the result that you are after. For example, you want to report on total time off available to employees. Your database design divides time off into separate buckets for vacation time and sick time. You however, wish to report a single value.

Solution

Write an expression involving the existing columns in the table, and then place the expression into your SELECT list. Place it there as you would any other column. Provide a column alias by which the program executing the query can reference the computed column. For example:

```
SELECT BusinessEntityID AS "EmployeeID",
   VacationHours + SickLeaveHours AS "AvailableTimeOff"
FROM HumanResources.Employee;
```

```
 EmployeeID AvailableTimeOff
----------- ----------------
          1              168
          2               21
          3               23
...
```

How It Works

You can specify any expression you like in the SELECT list, and the value of that expression will be returned as a column in the query results. Most of the time you'll be referring to at least one table column from such an expression, but there are actually useful expressions that can be written that stand alone, that do not take other columns as input.

Recipe 1-8 introduces column aliases. It's especially important to provide them for computed columns. That's because if you don't provide an alias for a computed column, one is not created for you, and thus there is no name by which to refer to the column, nor is there a name to place in the output heading when executing the query ad-hoc from Management Studio.

1-11. Negating a Search Condition

Problem

You are finding it easier to describe those rows that you do *not* want rather than those that you *do* want.

Solution

Describe the rows that you do not want. Then use the NOT operator to essentially reverse the description so that you get those rows that you do want. The NOT logical operator negates the expression that follows it. For example, you can retrieve all employees having a title of anything but "Ms." by executing the following query:

```
SELECT  Title, FirstName, LastName
FROM  Person.Person
WHERE NOT Title = 'Ms.';
```

```
Title    FirstName   LastName
-------- ----------- -----------
Mr.      Jossef      Goldberg
Mr.      Hung-Fu     Ting
...
Sr.      Humberto    Acevedo
Sra.     Pilar       Ackerman
...
```

How It Works

NOT specifies the reverse of a search condition, in this case specifying that only rows that don't have the Title equal to "Ms." be returned. You can apply the NOT operator to individual expressions in a WHERE clause. You can also apply it to a group of expressions. For example:

```
WHERE NOT (Title = 'Ms.' OR Title = 'Mr.')
```

Think in terms of finding all the rows having "Ms." or "Mr." and then returning everything else except those rows. The parentheses force evaluation of the OR condition first. Then all rows not meeting that condition are returned by the query.

1-12. Keeping the WHERE Clause Unambiguous

Problem

You are writing several expressions in a WHERE clause that are linked together using AND and OR, and sometimes NOT. You worry that future maintainers of your query will misconstrue your intentions.

Solution

Enclose expressions in parentheses to make clear your intent. For example:

```
SELECT Title, FirstName, LastName
FROM   Person.Person
WHERE  Title = 'Ms.' AND
       (FirstName = 'Catherine' OR
       LastName = 'Adams');
```

How It Works

You can write multiple operators (AND, OR, NOT) in a single WHERE clause, but it is important to make your intentions clear by properly embedding your ANDs and ORs in parentheses. The NOT operator takes precedence (is evaluated first) over AND. The AND operator takes precedence over the OR operator. Using both AND and OR operators in the same WHERE clause without parentheses can return unexpected results.

Consider the solution query and pretend for a moment that there are no parentheses. Is the intention to return results for all rows with a Title of "Ms.," and of those rows, only include those with a FirstName of Catherine or a LastName of Adams? Or did the query author wish to search for all people titled "Ms." with a FirstName of Catherine, as well as anyone with a LastName of Adams? The parentheses make the author's intentions crystal clear.

It is good practice to use parentheses to clarify exactly what rows should be returned. Even if you are fully conversant with the rules of operator precedence, those who come after you may not be. Make judicious use of parentheses to remove all doubt as to your intentions.

1-13. Testing for Existence

Problem

You want to know whether something is true, but you don't really care to see the data that proves it. For example, you want to know the answer to the following business question: "Are there really employees having more than 80 hours of sick time?"

Solution

One solution is to execute a query to return one row in the event that what you care about is true, and to return no rows otherwise. The following example returns the value 1 in the event of any employee having more than 80 hours of sick time:

```
SELECT TOP(1) 1
FROM HumanResources.Employee
WHERE SickLeaveHours > 80;
```

```
-----------

(0 row(s) affected)
```

Another approach is to write an EXISTS predicate. For example, and testing for 40 hours this time:

```
SELECT 1
WHERE  EXISTS (
   SELECT *
   FROM HumanResources.Employee
   WHERE SickLeaveHours > 40
);
```

```
-----------
          1

(1 row(s) affected)
```

How It Works

The first solution makes use of T-SQL's TOP(n) syntax to end the query when the first row is found matching the condition. No rows were found in the example. You will find one though, if you lower the hour threshold to 40. There are employees having more than 40 hours of sick time, but none that have more than 80 hours.

The second solution achieves the same result, but through an EXISTS predicate. The outer query returns the value 1 as a single row and column to indicate that rows exist for the query listed in the EXISTS predicate. Otherwise, the outer query returns no row at all.

Avoid an ORDER BY clause when testing for existence like this recipe shows. You want query execution to stop as soon as possible. You can solve a different type of problem by using ORDER BY in conjunction with TOP.

1-14. Specifying a Range of Values

Problem

You wish to specify a range of values as a search condition. For example, you are querying a table having a date column. You wish to return rows having dates only in a specified range of interest.

Solution

Write a predicate involving the BETWEEN operator. That operator allows you to specify a range of values, in this case date values. For example, to find sales orders placed between the dates 7/23/2005 and 7/24/2005:

```
SELECT SalesOrderID, ShipDate
FROM Sales.SalesOrderHeader
WHERE ShipDate BETWEEN '2005-07-23 00:00:00.0' AND '2005-07-24 23:59:59.0';
```

```
SalesOrderID ShipDate
------------ -----------------------
       43758 2005-07-23 00:00:00.000
       43759 2005-07-23 00:00:00.000
       43760 2005-07-23 00:00:00.000
       ...
```

How It Works

This recipe demonstrates the BETWEEN operator, which tests whether a column's value falls between two values that you specify. The value range is inclusive of the two endpoints.

Notice that we designated the specific time in hours, minutes, and seconds as well. The time-of-day defaults to 00:00:00, which is midnight at the start of a date. In this example, we wanted to include all of 7/24/2005. Thus, we specified the last possible second of that day.

However, there is an issue you must be aware of when using BETWEEN with date-time values: What if the shipment date is 2005-07-23 23:59:59.456? A safer approach is to test for dates being greater than or equal to the starting point, and less than the earliest time just after the end point. For example:

```
SELECT SalesOrderID, ShipDate
FROM Sales.SalesOrderHeader
WHERE ShipDate >= '2005-07-23' AND ShipDate < '2005-07-25';
```

This solution is safer, because it's trivial to specify the earliest possible time on the 25th, and then to test for ShipDate being less than that. It is not so easy to know the maximum possible fractional seconds value to specify for the BETWEEN approach. Should those be 59.997? 59.9999? 59.999999? How many nines? Don't waste time trying to figure that out. Just take the safer approach unless you are certain that your data never includes fractional seconds.

■ **Caution** You encounter the same issue with decimal digits when using BETWEEN with decimal and floating-point values as with date-time values.

1-15. Checking for Null Values

Problem

Some of the values in a column might be NULL. You wish to identify rows having or not having NULL values.

Solution

Make use of the IS NULL and IS NOT NULL tests to identify rows having or not having NULL values in a given column. For example, the following query returns any rows for which the value of the product's weight is unknown:

```
SELECT  ProductID, Name, Weight
FROM    Production.Product
WHERE   Weight IS NULL;
```

```
  ProductID Name                    Weight
----------- --------------------  ------
          1 Adjustable Race         NULL
          2 Bearing Ball            NULL
          3 BB Ball Bearing         NULL
          4 Headset Ball Bearings   NULL
...
```

How It Works

`NULL` values cannot be identified using operators such as `=` and `<>` that are designed to compare two values and return a `TRUE` or `FALSE` result. `NULL` actually indicates the *absence* of a value. For that reason, neither of the following predicates can be used to detect a `NULL` value:

- `Weight = NULL` yields the value `UNKNOWN`, which is neither `TRUE` nor `FALSE`
- `Weight <> NULL` also yields `UNKNOWN`

`IS NULL`, however, is specifically designed to return `TRUE` when a value is `NULL`. Likewise, the expression `IS NOT NULL` returns `TRUE` when a value is not `NULL`. Predicates involving `IS NULL` and `IS NOT NULL` enable you to filter for rows having or not having `NULL` values in one or more columns.

■ **Caution** Improper handling of nulls is one of the most prevalent sources of query mistakes. See Chapter 3 for guidance and techniques that can help you avoid trouble and get the results you want.

1-16. Writing an IN-List

Problem

You are searching for matches to a specific list of values. You could write a string of predicates joined by `OR` operators, but you prefer a more easily readable and maintainable solution.

Solution

Create a predicate involving the `IN` operator, which allows you to specify an arbitrary list of values. For example, the `IN` operator in the following query tests the equality of the `Color` column to a list of expressions:

```
SELECT  ProductID, Name, Color
FROM Production.Product
WHERE Color IN ('Silver', 'Black', 'Red');
```

```
  ProductID Name              Color
----------- ----------------- ---------
        317 LL Crankarm       Black
        318 ML Crankarm       Black
        319 HL Crankarm       Black
        320 Chainring Bolts   Silver
        321 Chainring Nut     Silver
...
```

How It Works

Use the IN operator any time you have a specific list of values. You can think of IN as shorthand for multiple OR expressions. For example, the following two WHERE clauses are semantically equivalent:

```
WHERE Color IN ('Silver', 'Black', 'Red')

WHERE Color = 'Silver' OR Color = 'Black' OR Color = 'Red'
```

You can see that an IN-list becomes less cumbersome than a string of OR'd-together expressions. This is especially true as the number of values grows. You can also write NOT IN to find rows having values other than those in your list.

■ **Caution** Take care when writing NOT IN. If just one value in the in-list is null, your NOT IN expression will always return UNKNOWN, and no rows will be selected. You won't have that problem when writing an in-list of literal values, such as in the example, but the problem can occur easily when your in-list is made up of variables or table columns.

1-17. Performing Wildcard Searches

Problem

You don't have a specific value or list of values to find. What you do have is a general pattern, and you want to find all values that match that pattern.

Solution

Make use of the LIKE predicate, which provides a set of basic pattern-matching capabilities. Create a string using so-called wildcards to serve as a search expression. Table 1-2 shows the wildcards available in SQL Server 2014.

Table 1-2. *Wildcards for the LIKE predicate*

Wildcard	Usage
`%`	The percent sign. Represents a string of zero or more characters
`_`	The underscore. Represents a single character
`[...]`	A list of characters enclosed within square brackets. Represents a single character from among any in the list.
`[^...]`	A list of characters enclosed within square brackets and preceded by a caret. Represents a single character from among any not in the list.

The following example demonstrates using the `LIKE` operation with the `%` wildcard, searching for any product with a name beginning with the letter B:

```
SELECT ProductID, Name
FROM Production.Product
WHERE Name LIKE 'B%';
```

This query returns the following results:

```
  ProductID Name
----------- ----------------------
          3 BB Ball Bearing
          2 Bearing Ball
        877 Bike Wash - Dissolver
        316 Blade
```

How It Works

Wildcards allow you to search for patterns in character-based columns. In the example from this recipe, the `%` sign is used to represent a string of zero or more characters:

```
WHERE Name LIKE 'B%'
```

If searching for a literal that would otherwise be interpreted by SQL Server as a wildcard, you can use the `ESCAPE` clause. For example, you can search for a literal percentage sign in the `Name` column:

```
WHERE Name LIKE '%/%%' ESCAPE '/'
```

A slash embedded in single quotes is put after the `ESCAPE` clause in this example. This designates the slash symbol as the escape character for the associated expression string. Any wildcard preceded by a slash is then treated as just a regular character.

■ **Tip** If you ever find yourself making extensive use of `LIKE`, especially in finding words or phrases within large text columns, be sure to become familiar with SQL Server's full-text search feature. *Pro Full-Text Search in SQL Server 2008* by Hilary Cotter and Michael Coles is a good resource on that feature and its use.

1-18. Sorting Your Results

Problem

You are executing a query, and you wish the results to come back in a specific order.

Solution

Write an ORDER BY clause into your query. Specify the columns on which to sort. Place the clause at the very end of your query. For example:

```
SELECT p.Name, h.EndDate, h.ListPrice
FROM   Production.Product p
INNER JOIN Production.ProductListPriceHistory h ON
           p.ProductID = h.ProductID
ORDER BY p.Name, h.EndDate;
```

This query returns results as follows:

```
Name                    EndDate                 ListPrice
----------------------- ----------------------- ---------
All-Purpose Bike Stand  NULL                       159.00
AWC Logo Cap            NULL                         8.99
AWC Logo Cap            2006-06-30 00:00:00.000    8.6442
AWC Logo Cap            2007-06-30 00:00:00.000    8.6442
Bike Wash - Dissolver   NULL                         7.95
Cable Lock              2007-06-30 00:00:00.000     25.00
...
```

Notice the results are first sorted by Name. Within Name, they are sorted by EndDate.

How It Works

Although queries sometimes appear to return data properly without an ORDER BY clause, you should never depend upon any ordering that is accidental. You must write an ORDER BY into your query if the order of the result set is critical. You can designate one or more columns in your ORDER BY clause, so long as the columns do not exceed 8,060 bytes in total.

We can't stress enough the importance of ORDER BY when order matters. Grouping operations and indexing sometimes make it seem that ORDER BY is superfluous. It isn't. Trust us: there are enough corner cases that sooner or later you'll be caught out. If the sort order matters, then say so explicitly in your query by writing an ORDER BY clause.

■ **Note** The solution query implements what is known as a *join* between two tables. There's a lot to be said about joins, and you'll learn more about them in Chapter 4.

The default sort order is an ascending sort. You can specify ascending or descending explicitly by writing either ASC and DESC, as follows:

```
ORDER BY p.Name ASC, h.EndDate DESC
```

NULL values are considered lower than everything else. They sort to the top in an ascending sort. They sort to the bottom in a descending sort.

You need not return a column in order to sort by it. For example, you can group results by color to help break any ties:

```
ORDER BY p.Name, h.EndDate, p.Color
```

It doesn't matter that Color is not returned by the query. SQL Server can sort on the column without returning it.

1-19. Specifying the Case-Sensitivity of a Sort

Problem

You want to specify whether a sort is performed in a binary manner, or whether it is case-sensitive or case-insensitive.

Solution

Add a COLLATE clause to each column specification in your ORDER BY clause that you are concerned about. Following is a repeat of the query from Recipe 1-18, but this time a binary sort is specified for the p.Name column.

```
SELECT p.Name, h.EndDate, h.ListPrice
FROM Production.Product p
INNER JOIN Production.ProductListPriceHistory h ON
      p.ProductID = h.ProductID
ORDER BY p.Name COLLATE Latin1_General_BIN ASC,
         h.EndDate DESC;
```

We've tampered with one of the product names in our copy of the Adventure Works database in order to demonstrate the effect of this query and its collation. Look at where frame size 42 occurs in the following output.

```
...
HL Headset                     2007-06-30 00:00:00.000             124.73
HL MOUNTAIN FRAME - BLACK, 42  2007-06-30 00:00:00.000           1226.9091
HL MOUNTAIN FRAME - BLACK, 42  2006-06-30 00:00:00.000           1191.1739
HL MOUNTAIN FRAME - BLACK, 42  NULL                                1349.60
HL Mountain Frame - Black, 38  2007-06-30 00:00:00.000           1226.9091
HL Mountain Frame - Black, 38  2006-06-30 00:00:00.000           1191.1739
HL Mountain Frame - Black, 38  NULL                                1349.60
HL Mountain Frame - Black, 44  2006-06-30 00:00:00.000             1349.60
...
```

The default collation in the example database produces a different result:

```
...
HL Headset                     2007-06-30 00:00:00.000             124.73
HL Mountain Frame - Black, 38  2007-06-30 00:00:00.000           1226.9091
HL Mountain Frame - Black, 38  2006-06-30 00:00:00.000           1191.1739
HL Mountain Frame - Black, 38  NULL                                1349.60
HL MOUNTAIN FRAME - BLACK, 42  2007-06-30 00:00:00.000           1226.9091
HL MOUNTAIN FRAME - BLACK, 42  2006-06-30 00:00:00.000           1191.1739
HL MOUNTAIN FRAME - BLACK, 42  NULL                                1349.60
HL Mountain Frame - Black, 44  2006-06-30 00:00:00.000             1349.60
...
```

How It Works

You have the option to specify a non-default collation for each column listed in an ORDER BY clause. You can of course explicitly specify the default collation too, but typically you would add a COLLATE clause because you want something other than the default.

SQL Server supports thousands of collations, each providing a different set of sorting rules. You can obtain a complete list by executing the following query:

```
SELECT Name, Description
FROM fn_helpcollations();
```

The list is long. It helps to narrow your search. You can get an idea as to the languages that are supported by executing the following query:

```
SELECT DISTINCT SUBSTRING(Name, 1, CHARINDEX('_', Name)-1)
FROM fn_helpcollations();
```

Then you can list the collations for just one language. For example, here is how to list collations for Ukrainian:

```
SELECT Name, Description
FROM fn_helpcollations()
WHERE Name LIKE 'Ukrainian%';
```

Each language's collation set generally provides the ability for you to choose whether any of the following matter when sorting rows: case, accents, kanatype, and character width. Kanatype matters for Japanese text. Character width comes into play in some situations in which Unicode provides the same character in, for example, single-byte or double-byte form.

A binary collation such as the Latin1_General_BIN used in the example is what you need in order to return a case-sensitive sort in the way that many programmers think of such as sort as being done. There is also a Latin1_General_CS_AS collation that is described as being case-sensitive and accent-sensitive. And that is true! But the sorting is done according to Unicode rules, and the results sometimes appear to match those from the insensitive Latin1_General_CI_AI.

Unicode sorting rules view uppercase as being greater than lowercase. Thus, "BLUE" sorts after "blue" when a case-sensitive sort is being performed. However "BLUE" will sort prior to "red" in either case, because Unicode rules only look at the case when it is needed in order to break a tie. If your column contains all distinct values such as "BLUE" and "Red," then they will sort the same no matter whether you use Latin1_General_CS_AS or Latin1_General_CI_AI, and that can be disconcerting at first. It is when you have values such as "reD," "Red," and "RED" that you will see a difference in results between case-sensitive and case-insensitive sorts done under Unicode sorting rules.

■ **Note** Visit http://www.unicode.org/reports/tr10/ to read about Unicode's collation algorithm in extreme detail.

1-20. Sorting Nulls High or Low

Problem

You are a refugee from Oracle Database, and you miss the ability to specify NULLS FIRST and NULLS LAST when writing ORDER BY clauses.

Solution

Add a semaphore expression to your ORDER BY clause for the column in question. Then specify ASC or DSC to make the nulls sort first or last as desired. The following example adds such an expression for the Weight column in order to sort that column with nulls last.

```
SELECT  ProductID, Name, Weight
FROM    Production.Product
ORDER BY ISNULL(Weight, 1) DESC, Weight;
```

```
  ProductID Name                  Weight
----------- --------------------- -------
        826 LL Road Rear Wheel    1050.00
        827 ML Road Rear Wheel    1000.00
        818 LL Road Front Wheel    900.00
...
        504 Cup-Shaped Race          NULL
        505 Cone-Shaped Race         NULL
        506 Reflector                NULL

(504 row(s) affected)
```

How It Works

SQL Server doesn't implement syntax for you to use in specifying whether nulls sort first or last. The solution works around that omission by evaluating the following expression during the sort:

```
ISNULL(Weight, 1)
```

A null weight yields a result of 1. Otherwise, the expression is itself null. Those are the only two possible results: 1 or null. It's a simple matter to then append ASC or DESC to specify whether the rows returning 1 sort last or first.

If you find it confusing to evaluate ISNULL in your head, then you can get the same effect through the IIF function:

```
SELECT  ProductID, Name, Weight
FROM    Production.Product
ORDER BY IIF(Weight IS NULL, 1, 0), Weight;
```

The result from IIF in this example is 1 for null and zero otherwise. The normal sort order is ascending. Rows causing the expression to evaluate to zero have non-null weights and are sorted first. The null weights trigger IIF to return a 1, and they sort last.

1-21. Forcing Unusual Sort Orders

Problem

You wish to force a sort order not directly supported by the data. For example, you wish to retrieve only the colored products, and you further wish to force the color red to sort first.

Solution

Write an expression to translate values in the data to values that will give the sort order you are after. Then order your query results by that expression. Following is one approach to the problem of retrieving colored parts and listing the red ones first:

```
SELECT p.ProductID, p.Name, p.Color
FROM Production.Product AS p
WHERE p.Color IS NOT NULL
ORDER BY CASE p.Color
WHEN 'Red' THEN NULL ELSE p.COLOR END;
```

```
  ProductID Name                          Color
----------- ----------------------------- -----
        706 HL Road Frame - Red, 58       Red
        707 Sport-100 Helmet, Red         Red
        725 LL Road Frame - Red, 44       Red
...
        790 Road-250 Red, 48              Red
        791 Road-250 Red, 52              Red
        792 Road-250 Red, 58              Red
        793 Road-250 Black, 44            Black
        794 Road-250 Black, 48            Black
        795 Road-250 Black, 52            Black
```

How It Works

The solution takes advantage of the fact that SQL Server sorts nulls first. The `CASE` expression returns `NULL` for red-colored items, thus forcing those first. Other colors are returned unchanged. The result is all the red items appear first in the list, and then red is followed by other colors in their natural sort order.

You don't have to rely upon nulls sorting first. You can translate "Red" to any value you like, such as, for example, to a single space character. Then that space character would sort before all the spelled-out color names.

1-22. Paging Through a Result Set

Problem

You wish to present an ordered result set to an application user N rows at a time.

Solution

Make use of the query-paging feature that was introduced in SQL Server 2012. Do this by adding `OFFSET` and `FETCH` clauses to your query's `ORDER BY` clause. For example, the following query uses `OFFSET` and `FETCH` to retrieve the first ten rows of results:

```
SELECT ProductID, Name
FROM Production.Product
ORDER BY Name
OFFSET 0 ROWS FETCH NEXT 10 ROWS ONLY;
```

Results from this query will be the first ten rows, as ordered by product name:

```
  ProductID Name
----------- ------------------------
          1 Adjustable Race
        879 All-Purpose Bike Stand
        712 AWC Logo Cap
          3 BB Ball Bearing
          2 Bearing Ball
        877 Bike Wash - Dissolver
        316 Blade
        843 Cable Lock
        952 Chain
        324 Chain Stays
```

Changing the offset from 0 to 8 will fetch another ten rows. The offset will skip the first eight rows. There will be a two-row overlap with the preceding result set. Here is the query:

```
SELECT ProductID, Name
FROM Production.Product
ORDER BY Name
OFFSET 8 ROWS FETCH NEXT 10 ROWS ONLY;
```

And here are the results:

```
  ProductID Name
----------- --------------------
        952 Chain
        324 Chain Stays
        322 Chainring
        320 Chainring Bolts
        321 Chainring Nut
        866 Classic Vest, L
        865 Classic Vest, M
        864 Classic Vest, S
        505 Cone-Shaped Race
        323 Crown Race
```

Continue modifying the offset each time, paging through the result until the user is finished.

How It Works

`OFFSET` and `FETCH` turn a `SELECT` statement into a query fetching a specific window of rows from those that are possible. Use `OFFSET` to specify how many rows to skip from the beginning of the possible result set. Use `FETCH` to set the number of rows to return. You can change either value as you wish from one execution to the next.

Be sure to specify a deterministic set of sort columns in your `ORDER BY` clause. Each `SELECT` to get the next page of results is a separate query and a separate sort operation. Make sure that your data sorts the same way each time. Do not leave ambiguity.

■ **Note** The word *deterministic* means that the same inputs always give the same outputs. Specify your sort such that the same set of input rows will always yield the same ordering in the query output.

Each execution of a paging query is a separate execution from the others. Consider executing sequences of paging queries from within a transaction providing snapshot or serializable isolation. Chapter 12 discusses such transactions in detail. However, you can begin and end such a transaction as follows:

```
SET TRANSACTION ISOLATION LEVEL SNAPSHOT;
BEGIN TRANSACTION;
   /* Queries go here */
COMMIT;
/* Return to default */
SET TRANSACTION ISOLATION LEVEL READ COMMITTED;
```

Anomalies are possible without isolation. For example:

- You might see a row twice. In the solution example, if another user inserted eight new rows with names sorting earlier than "Adjustable Race," then the second query results would be the same as the first.
- You might miss rows. If another user quickly deleted the first eight rows, then the second solution query would miss everything from "Chainring" to "Crown Race."

You may decide to risk the default isolation level. If your target table is read-only, or if it is updated in batch-mode only at night, then you might be justified in leaving the isolation level at its default because the risk of change during the day is low to non-existent. Possibly you might choose not to worry about the issue at all. However, make sure that whatever you do is the result of thinking things through and making a conscious choice.

■ **Note** It may seem rash for us to even hint at not allowing the possibility of inconsistent results. We advocate making careful and conscious decisions. Some applications—Facebook is a well-known example—trade away some consistency in favor of performance. (We routinely see minor inconsistencies on our Facebook walls). We are not saying you should do the same. We simply acknowledge the possibility of such a choice.

1-23. Sampling a Subset of Rows

Problem

You are getting familiar with a table, and you want to review a representative sampling of the data.

Solution

Query the table and limit the results using the `TABLESAMPLE` clause. You can specify an approximate percentage of rows to retrieve:

```
SELECT *
FROM Purchasing.PurchaseOrderHeader
TABLESAMPLE (5 PERCENT);
```

Or you can specify an approximate quantity of rows:

```
SELECT *
FROM Purchasing.PurchaseOrderHeader
TABLESAMPLE (200 ROWS);
```

How It Works

The `TABLESAMPLE` clause is available from SQL Server 2008 R2 forward. Use it to get an idea of what the data looks like in a table without having to page through all of the table's data.

The values specified for rows and percentages should be thought of as approximate values. If you specify a low enough value for rows, such as 20 rows in the example queries, you might not get any data back at all. That's because at some point during processing, the number or percentage of rows you specify is translated into some integer number of data pages relative to all the pages that are allocated to the table. That number of pages is randomly chosen from among all the pages, and all rows that happen to be on the selected pages are returned. The actual distribution of rows across the pages can affect the results, as can the rounding to an integer number of pages.