**Chap te r 3**

Filtering and Sorting Data

Exam objectives in this chapter:

■■ Work with Data

■■ Query data by using SELECT statements.

■■ Implement data types.

■■ Modify Data

■■ Work with functions.

Filtering and sorting data are the most foundational, as well as most common, aspects of

querying data. Almost every query that you write needs to filter data, and many queries

involve sorting. The traditional way to filter data in T-SQL is based on predicates. However,

T-SQL also supports filtering data based on another concept—a specified number of rows

and ordering. The options T-SQL supports based on this concept are TOP and OFFSETFETCH.

As for sorting, even though it might seem like a trivial aspect of querying, it’s actually

a source for quite a lot of confusion and misunderstanding, which this chapter tries to

clarify.

Lessons in this chapter:

■■ Lesson 1: Filtering Data with Predicates

■■ Lesson 2: Sorting Data

■■ Lesson 3: Filtering Data with TOP and OFFSET-FETCH

**Before You Begin**

To complete the lessons in this this chapter, you must have:

■■ Experience working with Microsoft SQL Server Management Studio (SSMS).

■■ Some experience writing T-SQL code.

■■ Access to a SQL Server 2012 instance with the sample database TSQL2012 installed.

**Lesson 1: Filtering Data with Predicates**

T-SQL supports three query clauses that enable you to filter data based on predicates. Those

are the ON, WHERE, and HAVING clauses. The ON and HAVING clauses are covered later in

the book. ON is covered as part of the discussions about joins in Chapter 4, “Combining Sets,”

and HAVING is covered as part of the discussions about grouping data in Chapter 5, “Grouping

and Windowing.” Lesson 1 in this chapter focuses on filtering data with the WHERE clause.

After this lesson, you will be able to:

■■ Use the WHERE clause to filter data based on predicates.

■■ Filter data involving NULLs correctly.

■■ Use search arguments to filter data efficiently.

■■ Combine predicates with logical operators.

■■ Understand the implications of three-valued logic on filtering data.

■■ Filter character data.

■■ Filter date and time data.

Estimated lesson time: 60 minutes

Predicates, Three-Valued Logic, and Search Arguments

In the very first SQL queries that you ever wrote, you very likely already started using the

WHERE clause to filter data based on predicates. Initially, it seems like a very simple and

straightforward concept. But with time, as you gain deeper understanding of T-SQL, you probably

realize that there are filtering aspects that are not that obvious. For example, you need

to understand how predicates interact with NULLs, and how filters based on such predicates

behave. You also need to understand how to form your predicates to maximize the efficiency

of your queries, and for this you need to be familiar with the concept of a *search argument*.

Some of the examples in this chapter use the HR.Employees table from the TSQL2012

sample database. Here’s the content of the table (only relevant columns shown).

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 Buck UK NULL London

6 Paul Suurs UK NULL London

7 Russell King UK NULL London

8 Maria Cameron USA WA Seattle

9 Zoya Dolgopyatova UK NULL London

To start with a simple example, consider the following query, which filters only employees

from the United States.

SELECT empid, firstname, lastname, country, region, city

FROM HR.Employees

WHERE country = N'USA';

Recall from Chapter 1, “Foundations of Querying,” that a predicate is a logical expression.

When NULLs are not possible in the data (in this case, the country column is defined as not

allowing NULLs), the predicate can evaluate to true or false. The type of logic used in such a

case is known as *two-valued logic*. The WHERE filter returns only the rows for which the predicate

evaluates to true. Here’s the result of this query.

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. Consider the customer

location columns country, region, and city in the Sales.Customers table. Suppose that these

columns reflect the location hierarchy based on the sales organization. For some places in the

world, such as in the United States, all three location columns are applicable; for example:

Country: USA

Region: WA

City: Seattle

But other places, like the United Kingdom, have only two applicable parts—the country

and the city. In such cases, the region column is set to NULL; for example:

Country: UK

Region: NULL

City: London

Consider then a query filtering only employees from Washington State.

SELECT empid, firstname, lastname, country, region, city

FROM HR.Employees

WHERE region = N'WA';

Recall from Chapter 1 that when NULLs are possible in the data, a predicate can evaluate

to true, false, and unknown. This type of logic is known as *three-valued logic*. When using an

equality operator in the predicate like in the previous query, you get true when both operands

are not NULL and equal; for example, WA and WA. You get false when both are not

NULL and different; for example, OR and WA. So far, it’s straightforward. The tricky part is

when NULL marks are involved. You get an unknown when at least one operand is NULL; for

example, NULL and WA, or even NULL and NULL.

As mentioned, the WHERE filter returns rows for which the predicate evaluates to true,

meaning that it discards both false and unknown cases. Therefore, the query returns only

employees where the region is not NULL and equal to WA, as shown in the following.

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

You might consider this behavior as intuitive, but consider a request to return only employees

that are not from Washington State. You issue the following query.

SELECT empid, firstname, lastname, country, region, city

FROM HR.Employees

WHERE region <> N'WA';

Run the query and you get an empty set back:

empid firstname lastname country region city

------ ---------- --------- -------- ------- ---------

Can you make sense of the result?

As it turns out, all of the employees that aren’t from Washington State are from the UK;

recall that the region for places in the UK is set to NULL to indicate that it’s inapplicable. Even

though it may be clear to you that someone from the UK isn’t from Washington State, it’s

not clear to T-SQL. To T-SQL, a NULL represents a missing value that could be applicable, and

could be WA just like it could be anything else. So it cannot conclude with certainty that the

region is different from WA. In other words, when region is NULL, the predicate region <>

'WA' evaluates to unknown, and the row is discarded. So such a predicate would return only

cases that are not NULL and are known to be different from WA. For example, if you had an

employee in the table with a region NY, such an employee would have been returned.

Knowing that in the Employees table a NULL region represents a missing and inapplicable

region, how do you make T-SQL return such employees when looking for places where the

region is different from WA?

If you’re considering a predicate such as region <> N'WA' OR region = NULL, you need to

remember that two NULLs are not considered equal to each other. The result of the expression

NULL = NULL is, in fact, unknown—not true. T-SQL provides the predicate IS NULL to

return a true when the tested operand is NULL. Similarly, the predicate IS NOT NULL returns

true when the tested operand is not NULL. So the solution to this problem is to use the following

form.

SELECT empid, firstname, lastname, country, region, city

FROM HR.Employees

WHERE region <> N'WA'

OR region IS NULL;

Here’s the result of this query.

empid firstname lastname country region city

------ ---------- ------------- -------- ------- -------

5 Sven Buck UK NULL London

6 Paul Suurs UK NULL London

7 Russell King UK NULL London

9 Zoya Dolgopyatova UK NULL London

Query filters have an important performance side to them. For one thing, by filtering

rows in the query (as opposed to in the client), you reduce network traffic. Also, based on

the query filters that appear in the query, SQL Server can evaluate the option to use indexes

to get to the data efficiently without requiring a full scan of the table. It’s important to note,

though, that the predicate needs to be of a form known as a *search argument* (SARG) to allow

efficient use of the index. Chapter 15, “Implementing Indexes and Statistics,” goes into details

about indexing and the use of search arguments; here, I’ll just briefly describe the concept

and provide simple examples.

A predicate in the form *column operator value* or *value operator column* can be a search

argument. For example, predicates like col1 = 10, and col1 > 10 are search arguments. Applying

manipulation to the filtered column in most cases prevents the predicate from being a

search argument. An example for manipulation of the filtered column is applying a function

to it, as in F(col1) = 10, where F is some function. There are some exceptions to this rule, but

they are very uncommon.

For example, suppose you have a stored procedure that accepts an input parameter @dt

representing an input shipped date. The procedure is supposed to return orders that were

shipped on the input date. If the shippeddate column did not allow NULLs, you could use the

following query to address this task.

SELECT orderid, orderdate, empid

FROM Sales.Orders

WHERE shippeddate = @dt;

However, the shippeddate column does allow NULLs; those represent orders that weren’t

shipped yet. When users will need all orders that were not shipped yet, the users will provide

a NULL as the input shipped date, and your query would need to be able to cope with such a

case. Remember that when comparing two NULLs, you get unknown and the row is filtered

out. So the current form of the predicate doesn’t address NULL inputs correctly. Some address

this need by using COALESCE or ISNULL to substitute NULLs with a value that doesn’t exist in

the data normally, as in the following.

SELECT orderid, orderdate, empid

FROM Sales.Orders

WHERE COALESCE(shippeddate, '19000101') = COALESCE(@dt, '19000101');

The problem is that even though the solution now returns the correct result—even when

the input is NULL—the predicate isn’t a search argument. This means that SQL Server cannot

efficiently use an index on the shippeddate column. To make the predicate a search argument,

you need to avoid manipulating the filtered column and rewrite the predicate like the

following.

SELECT orderid, orderdate, empid

FROM Sales.Orders

WHERE shippeddate = @dt

OR (shippeddate IS NULL AND @dt IS NULL);

*Exam Tip*

Understanding the impact of using COALESCE and ISNULL on performance is an important

skill for the exam.

Interestingly, standard SQL has a predicate called IS NOT DISTINCT FROM that has the

same meaning as the predicate used in the last query (return true when both sides are equal

or when both are NULLs, otherwise false). Unfortunately, T-SQL doesn’t support this predicate.

Another example for manipulation involves the filtered column in an expression; for

example, col1 - 1 <= @n. Sometimes, you can rewrite the predicate to a form that is a search

argument, and then allow efficient use of indexing. The last predicate, for example, can be

rewritten using simple math as col1 <= @n + 1.

In short, when a predicate involves manipulation of the filtered column, and there are

alternative ways to phrase it without the manipulation, you can increase the likelihood for

efficient use of indexing. There are a couple of additional examples in the sections “Filtering

Character Data” and “Filtering Date and Time Data” later in this chapter. And as mentioned,

more extensive coverage of the topic is in Chapter 15.

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 earlier in this chapter the query that returned all employees from Washington

State; 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 from Chapter 1 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 like the following.

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.

As discussed in Chapter 2, “Getting Started with the SELECT Statement,” 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 very 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 may be cases

where implicit conversion hurts performance. It is a best practice to use the proper form, like

in the following.

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 3-1 describes

the available wildcards, their meaning, and an example demonstrating their use.

**Table 3-1** 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 Zoya Dolgopyatova

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.

*IMPORTANT* **Performance of the LIKE Predicate**

When the LIKE pattern starts with a known prefix—for example, col LIKE 'ABC%'—

SQL Server can potentially efficiently use an index on the filtered column; in other words,

SQL Server can rely on index ordering. When the pattern starts with a wildcard—for example,

col LIKE '%ABC%'—SQL Server cannot rely on index ordering anymore. Also, when

looking for a string that starts with a known prefix (say, ABC) make sure you use the LIKE

predicate, as in col LIKE 'ABC%', because this form is considered a search argument. Recall

that applying manipulation to the filtered column prevents the predicate from being a

search argument. For example, the form LEFT(col, 3) = 'ABC' isn’t a search argument and

will prevent SQL Server from being able to use an index efficiently.

Filtering Date and Time Data

There are several important considerations when filtering date and time data that are related

to both the correctness of your code and to its performance. You want to think of things like

how to express literals, filter ranges, and use search arguments.

I’ll start with literals. Suppose that you need to query the Sales.Orders table and return

only orders placed on February 12, 2007. You use the following query.

SELECT orderid, orderdate, empid, custid

FROM Sales.Orders

WHERE orderdate = '02/12/07';

If you’re an American, this form probably means February 12, 2007, to you. However, if

you’re British, this form probably means December 2, 2007. If you’re Japanese, it probably

means December 7, 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 logon that runs the code. Each logon has a

default language associated with it, and the default language sets various session options on

the logon’s behalf, including one called DATEFORMAT. A logon 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 '20070212' is always interpreted as ymd, regardless of your language.

Note that the form '2007-02-12' 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 like the following.

SELECT orderid, orderdate, empid, custid

FROM Sales.Orders

WHERE orderdate = '20070212';

*Note* **Storing Dates in a DATETI ME Column**

The filtered column orderdate is of a DATETIME data type representing both date and time.

Yet the literal specified in the filter contains only a date part. When SQL Server converts the

literal to the filtered column’s type, it assumes midnight when a time part isn’t indicated.

If you want such a filter to return all rows from the specified date, you need to ensure that

you store all values with midnight as the time.

Another approach is to use the CONVERT or PARSE functions, which you can use to indicate

how you want SQL Server to interpret the literal that you specify. The CONVERT function

supports a style number representing the conversion style, and the PARSE function supports

indicating a culture name. You can find details about both functions in Chapter 2.

Another important aspect of filtering date and time data is trying whenever possible to

use search arguments. For example, suppose that you need to filter only orders placed in

February 2007. You can use the YEAR and MONTH functions, as in the following.

SELECT orderid, orderdate, empid, custid

FROM Sales.Orders

WHERE YEAR(orderdate) = 2007 AND MONTH(orderdate) = 2;

However, because here you apply manipulation to the filtered column, the predicate is

not considered a search argument, and therefore, SQL Server won’t be able to rely on index

ordering. You could revise your predicate as a range, like the following.

SELECT orderid, orderdate, empid, custid

FROM Sales.Orders

WHERE orderdate >= '20070201' AND orderdate < '20070301';

Now that you don’t apply manipulation to the filtered column, the predicate is considered

a search argument, and there’s the potential for SQL Server to rely on index ordering.

If you’re wondering why this code expresses the date range by using greater than or equal

to (>=) and less than (<) operators as opposed to using BETWEEN, there’s a reason for this.

When you are using BETWEEN and the column holds both date and time elements, what do

you use as the end value? As you might realize, for different types, there are different precisions.

What’s more, suppose that the type is DATETIME, and you use the following predicate.

WHERE orderdate BETWEEN '20070201' AND '20070228 23:59:59.999'

This type’s precision is three and a third milliseconds. The milliseconds part of the end

point 999 is not a multiplication of the precision unit, so SQL Server ends up rounding the value

to midnight of March 1, 2007. As a result, you may end up getting some orders that you’re

not supposed to see. In short, instead of BETWEEN, use >= and <, and this form will work correctly

in all cases, with all date and time types, whether the time portion is applicable or not.

**Quick Check**

1. What are the performance benefits in using the WHERE filter?

2. What is the form of a filter predicate that can rely on index ordering called?

**Quick Check Answer**

1. You reduce network traffic by filtering in the database server instead of in

the client, and you can potentially use indexes to avoid full scans of the tables

involved.

2. A search argument, or SARG, for short.

Practice **Filtering Data with Predicates**

In this practice, you exercise your knowledge of filtering data with predicates.

If you encounter a problem completing an exercise, you can install the completed projects

from the Solution folder that is provided with the companion content for this chapter and

lesson.

Exercise 1 Use the WHERE Clause to Filter Rows with NULLs

In this exercise, you practice the use of the WHERE clause to filter unshipped orders from the

Sales.Orders table.

1. Open SSMS and connect to the sample database TSQL2012.

2. You are asked to write a query that returns orders that were not shipped yet. Such

orders have a NULL in the shippeddate column. For your first attempt, use the following

query.

SELECT orderid, orderdate, custid, empid

FROM Sales.Orders

WHERE shippeddate = NULL;

However, when you run this code, you get an empty result set.

orderid orderdate custid empid

----------- ----------------------- ----------- -----------

The reason for this is that when the expression compares two NULLs, the result is unknown,

and the row is filtered out.

3. Revise the filter predicate to use the IS NULL operator instead of e quality (=), as in t he

following.

SELECT orderid, orderdate, custid, empid

FROM Sales.Orders

WHERE shippeddate IS NULL;

This time, you do get the correct result, shown here in abbreviated form.

orderid orderdate custid empid

----------- ----------------------- ----------- -----------

11008 2008-04-08 00:00:00.000 20 7

11019 2008-04-13 00:00:00.000 64 6

11039 2008-04-21 00:00:00.000 47 1

Exercise 2 Use the WHERE Clause to Filter a Range of Dates

In this exercise, you practice the use of the WHERE clause to filter orders within a certain

range of dates from the Sales.Orders table.

1. You are requested to return all orders that were placed between February 11, 2008,

and February 12, 2008. The orderdate column you’re supposed to filter by is of a

DATETIME type. With the current data in the table, all orderdate values have the time

set to midnight, but suppose this wasn’t the case—namely, that the time portion could

be a value other than midnight. For your first attempt, use the BETWEEN predicate, as

follows.

SELECT orderid, orderdate, custid, empid

FROM Sales.Orders

WHERE orderdate BETWEEN '20080211' AND '20080212 23:59:59.999';

Because 999 is not a multiplication of the DATETIME type’s precision unit (three and a

third milliseconds), the end value in the range gets rounded to the next midnight, and

the result includes rows from February 13 that you didn’t ask for.

orderid orderdate custid empid

----------- ----------------------- ----------- -----------

10881 2008-02-11 00:00:00.000 12 4

10887 2008-02-13 00:00:00.000 29 8

10886 2008-02-13 00:00:00.000 34 1

10884 2008-02-12 00:00:00.000 45 4

10883 2008-02-12 00:00:00.000 48 8

10882 2008-02-11 00:00:00.000 71 4

10885 2008-02-12 00:00:00.000 76 6

2. To fix the problem, revise the range filter to use the >= and < operators, as follows.

SELECT orderid, orderdate, custid, empid

FROM Sales.Orders

WHERE orderdate >= '20080211' AND orderdate < '20080213';

This time, you get the correct result.

Lesson Summary

■■ With the WHERE clause, you can filter data by using predicates. Predicates in T-SQL use

three-valued logic. The WHERE clause returns cases where the predicate evaluates to

true and discards the rest.

■■ Filtering data by using the WHERE clause helps reduce network traffic and can potentially

enable using indexes to minimize I/O. It is important to try and phrase your

predicates as search arguments to enable efficient use of indexes.

■■ When filtering different types of data, like character and date and time data, it is

important to be familiar with best practices that will ensure that you write both correct

and efficient code.

Lesson Review

Answer the following questions to test your knowledge of the information in this lesson. You

can find the answers to these questions and explanations of why each answer choice is correct

or incorrect in the “Answers” section at the end of this chapter.

1. What does the term three-valued logic refer to in T-SQL?

A. The three possible logical result values of a predicate : true, false, and NULL

B. The three possible logical result values of a predicate : true, false, and unknown

C. The three possible logical result values of a predicate : 1, 0, and NULL

D. The three possible logical result values of a predicate : -1, 0, and 1

2. Which of the following literals are language-dependent for the DATETIME data type?

(Choose all that apply.)

A. '2012-02-12'

B. '02/12/2012'

C. '12/02/2012'

D. '20120212'

3. Which of the following predicates are search arguments? (Choose all that apply.)

A. DAY(orderdate) = 1

B. companyname LIKE 'A%'

C. companyname LIKE '%A%'

D. companyname LIKE '%A'

E. orderdate > = '20120212' AND orderdate < '20120213'

**Lesson 2: Sorting Data**

Sorting data is supposed to be a trivial thing, but as it turns out, it’s a source of a lot of confusion

in T-SQL. This lesson describes the critical difference in T-SQL between unsorted and

sorted data. It then describes the tools T-SQL provides you to sort data.

After this lesson, you will be able to:

■■ Use the ORDER BY clause to determine the order of rows in the result of a

query.

■■ Describe the difference between a query with and without an ORDER BY

clause.

■■ Control ascending and descending direction of ordering.

■■ Follow ordering best practices.

■■ Identify ordering restrictions when DISTINCT is used.

■■ Order by aliases that were assigned in the SELECT clause.

Estimated lesson time: 30 minutes

Understanding When Order Is Guaranteed

Probably one of the most confusing aspects of working with T-SQL is understanding when

a query result is guaranteed to be returned in particular order 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.

If you recall from Chapter 1, 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 may—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 CPUs 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 may 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, and that’s the focus of the next section.

Using the ORDER BY Clause to Sort Data

The only way to truly guarantee that the rows are returned from a query in a certain order is

by adding an ORDER BY clause.

For example, if you want to return information about employees from Washington State

in the United States, 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 header 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 those, 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, you can order

the result rows by elements that are not part of the SELECT list, as long as the result rows

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 oneto-

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 2006-07-23 00:00:00.000

10263 2006-07-31 00:00:00.000

10351 2006-11-20 00:00:00.000

...

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.

Indexing is discussed later in the Training Kit, but for now, suffice it to say that 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.

**Quick Check**

1. How do you guarantee the order of the rows in the result of a query?

2. What is the difference between the result of a query with and one without an

ORDER BY clause?

**Quick Check Answer**

1. The only way to do so is by adding an ORDER BY clause.

2. Without an ORDER BY clause, the result is relational (from an ordering perspective);

with an ORDER BY clause, the result is conceptually what the standard

calls a cursor.

Practice **Sorting Data**

In this practice, you exercise your knowledge of sorting data with the ORDER BY clause.

If you encounter a problem completing an exercise, you can install the completed projects

from the Solution folder that is provided with the companion content for this chapter and

lesson.

Exercise 1 Use the ORDER BY Clause with Nondeterministic Ordering

In this exercise, you practice using the ORDER BY clause to sort data, practicing nondeterministic

ordering.

1. Open SSMS and connect to the sample database TSQL2012.

2. You are asked to write a query that returns the orders for customer 77. Use the following

query.

SELECT orderid, empid, shipperid, shippeddate

FROM Sales.Orders

WHERE custid = 77;

You get the following result set.

orderid empid shipperid shippeddate

-------- ------ ---------- -----------------------

10992 1 3 2008-04-03 00:00:00.000

10805 2 3 2008-01-09 00:00:00.000

10708 6 2 2007-11-05 00:00:00.000

10310 8 2 2006-09-27 00:00:00.000

Note that because you didn’t specify an ORDER BY clause, there’s no assurance that the

rows will be returned in the order shown in the previous code. The only assurance that

you have is that you will get this particular set of rows.

3. You are asked to revise your query such that the rows will be sorted by shipperid. Add

an ORDER BY clause, as follows.

SELECT orderid, empid, shipperid, shippeddate

FROM Sales.Orders

WHERE custid = 77

ORDER BY shipperid;

The query now returns the following result.

orderid empid shipperid shippeddate

-------- ------ ---------- -----------------------

10708 6 2 2007-11-05 00:00:00.000

10310 8 2 2006-09-27 00:00:00.000

10992 1 3 2008-04-03 00:00:00.000

10805 2 3 2008-01-09 00:00:00.000

Now you guarantee that the rows will be returned by shipperid ordering, but is the

ordering deterministic? For example, can you tell with certainty what will be the order

among rows with the same shipper ID? The answer is no.

Exercise 2 Use the ORDER BY Clause with Deterministic Ordering

In this exercise, you practice using the ORDER BY clause to sort data, practicing deterministic

ordering.

1. You start this step with the query you wrote in step 3 of Exercise 1. You are given a

requirement to add secondary ordering by shipped date, descending. Add shipperid

DESC to the ORDER BY clause, as follows.

SELECT orderid, empid, shipperid, shippeddate

FROM Sales.Orders

WHERE custid = 77

ORDER BY shipperid, shippeddate DESC;

The query now returns the following result.

orderid empid shipperid shippeddate

-------- ------ ---------- -----------------------

10708 6 2 2007-11-05 00:00:00.000

10310 8 2 2006-09-27 00:00:00.000

10992 1 3 2008-04-03 00:00:00.000

10805 2 3 2008-01-09 00:00:00.000

Unlike in step 3, now it’s guaranteed that the rows with the same shipper ID will be

sorted by shipped date, descending. Is ordering now deterministic? Can you tell with

certainty what will be the order among rows with the same shipper ID and shipped

date? The answer is still no, because the combination of columns shipperid and

shippeddate isn’t unique, never mind what the current values that you see in the table

might lead you to think. Technically, there could be multiple rows in the result of this

query with the same shipperid and shippeddate values.

2. You are asked to revise the query from step 1 by guaranteeing deterministic ordering.

You need to define a tiebreaker. For example, define orderid DESC as a tiebreaker, as

follows.

SELECT orderid, empid, shipperid, shippeddate

FROM Sales.Orders

WHERE custid = 77

ORDER BY shipperid, shippeddate DESC, orderid DESC;

Now, in case of ties in the shipperid and shippeddate values, the row with the greater

orderid value will be sorted first.

Lesson Summary

■■ Queries normally return a relational result where ordering isn’t guaranteed. If you need

to guarantee presentation ordering, you need to add an ORDER BY clause to your query.

■■ With the ORDER BY clause, you can specify a list of expressions for primary ordering,

secondary ordering, and so on. With each expression, you can indicate ASC or DESC

for ascending or descending ordering, with ascending being the default.

■■ Even when an ORDER BY clause is specified, the result could still have nondeterministic

ordering. For deterministic ordering, the ORDER BY list must be unique.

■■ You can use ordinal positions of expressions from the SELECT list in the ORDER BY

clause, but this is considered a bad practice.

■■ You can sort by elements that do not appear in the SELECT list unless the DISTINCT

clause is also specified.

■■ Because the ORDER BY clause is conceptually evaluated after the SELECT clause, you

can refer to aliases assigned in the SELECT clause within the ORDER BY clause.

■■ For sorting purposes, SQL Server considers NULLs as being lower than non-NULL

marks and equal to each other. This means that when ascending ordering is used,

they sort together before non-NULL marks.

Lesson Review

Answer the following questions to test your knowledge of the information in this lesson. You

can find the answers to these questions and explanations of why each answer choice is correct

or incorrect in the “Answers” section at the end of this chapter.

1. When a query doesn’t have an ORDER BY clause, what is the order in which the rows

are returned?

A. Arbitrary order

B. Primary key order

C. Clustered index order

D. Insertion order

2. You want result rows to be sorted by orderdate descending, and then by orderid,

descending. Which of the following clauses gives you what you want?

A. ORDER BY orderdate, orderid DESC

B. ORDER BY DESC orderdate, DESC orderid

C. ORDER BY orderdate DESC, orderid DESC

D. DESC ORDER BY orderdate, orderid

3. You want result rows to be sorted by orderdate ascending, and then by orderid,

ascending. Which of the following clauses gives you what you want? (Choose all

that apply.)

A. ORDER BY ASC(orderdate, orderid)

B. ORDER BY orderdate, orderid ASC

C. ORDER BY orderdate ASC, orderid ASC

D. ORDER BY orderdate, orderid

**Lesson 3: Filtering Data with TOP and OFFSET -FETCH**

The first lesson covered filtering data by using predicates, and the second covered sorting

data. This third lesson in a sense mixes filtering and sorting concepts. Often, you need to filter

data based on given ordering and a specified number of rows. Think about requests such as

“return the three most recent orders” and “return the five most expensive products.” The filter

involves some ordering specification and a requested number of rows. T-SQL provides two

options to handle such filtering needs: one is the proprietary TOP option and the other is the

standard OFFSET-FETCH option that was introduced in SQL Server 2012.

After this lesson, you will be able to:

■■ Filter data by using the TOP option.

■■ Filter data by using the OFFSET-FETCH option.

Estimated lesson time: 45 minutes

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 (BIGINT data type). 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 2008-05-06 00:00:00.000 65 1

11076 2008-05-06 00:00:00.000 9 4

11075 2008-05-06 00:00:00.000 68 8

*Note* **TOP and Parentheses**

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 puts a ceiling on 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

----------- ----------------------- ----------- -----------

11076 2008-05-06 00:00:00.000 9 4

11077 2008-05-06 00:00:00.000 65 1

11075 2008-05-06 00:00:00.000 68 8

11074 2008-05-06 00:00:00.000 73 7

11070 2008-05-05 00:00:00.000 44 2

11071 2008-05-05 00:00:00.000 46 1

11073 2008-05-05 00:00:00.000 58 2

11072 2008-05-05 00:00:00.000 20 4

11067 2008-05-04 00:00:00.000 17 1

The TOP option isn’t limited to a constant input; instead, it allows you to specify a selfcontained

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

----------- ----------------------- ----------- -----------

11076 2008-05-06 00:00:00.000 9 4

11077 2008-05-06 00:00:00.000 65 1

11075 2008-05-06 00:00:00.000 68 8

11074 2008-05-06 00:00:00.000 73 7

11070 2008-05-05 00:00:00.000 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 optimization. For example, this query gave

the following output on one system.

orderid orderdate custid empid

----------- ----------------------- ----------- -----------

11011 2008-04-09 00:00:00.000 1 3

10952 2008-03-16 00:00:00.000 1 1

10835 2008-01-15 00:00:00.000 1 1

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 this

query was run, the system returned the following output.

orderid orderdate custid empid

----------- ----------------------- ----------- -----------

11077 2008-05-06 00:00:00.000 65 1

11076 2008-05-06 00:00:00.000 9 4

11075 2008-05-06 00:00:00.000 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

----------- ----------------------- ----------- -----------

11077 2008-05-06 00:00:00.000 65 1

11076 2008-05-06 00:00:00.000 9 4

11075 2008-05-06 00:00:00.000 68 8

11074 2008-05-06 00:00:00.000 73 7

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 the row with the greater order ID to “win.” To do so, add orderid DESC to your ORDER

BY clause, as follows.

SELECT TOP (3) WITH TIES orderid, orderdate, custid, empid

FROM Sales.Orders

ORDER BY orderdate DESC, orderid DESC;

Here’s the output of this query.

orderid orderdate custid empid

----------- ----------------------- ----------- -----------

11077 2008-05-06 00:00:00.000 65 1

11076 2008-05-06 00:00:00.000 9 4

11075 2008-05-06 00:00:00.000 68 8

The query is now deterministic, and the results are guaranteed to be repeatable, as long as

the underlying data doesn’t change.

To conclude this section, we’d just like to note that the TOP option can also be used in

modification statements to limit how many rows get modified, but modifications are covered

later in this Training Kit.

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 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 2008-04-16 00:00:00.000 10 1

11026 2008-04-15 00:00:00.000 27 4

...

11004 2008-04-07 00:00:00.000 50 3

11003 2008-04-06 00:00:00.000 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.

While in T-SQL, a FETCH clause requires an OFFSET clause, and 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;

Here’s an abbreviated form of the output.

orderid orderdate custid empid

----------- ----------------------- ----------- -----------

11027 2008-04-16 00:00:00.000 10 1

11026 2008-04-15 00:00:00.000 27 4

...

10249 2006-07-05 00:00:00.000 79 6

10248 2006-07-04 00:00:00.000 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 one system returned after

running the code.

orderid orderdate custid empid

----------- ----------------------- ----------- -----------

11011 2008-04-09 00:00:00.000 1 3

10952 2008-03-16 00:00:00.000 1 1

10835 2008-01-15 00:00:00.000 1 1

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 concept where you return to the user one page of rows at a time.

The user passes as input parameters to your procedure or a 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 (@pagesize - 1) \* @pagesize ROWS FETCH NEXT @pagesize ROWS ONLY;

With these inputs, the code returns the following output.

orderid orderdate custid empid

----------- ----------------------- ----------- -----------

10477 2007-03-17 00:00:00.000 60 5

10476 2007-03-17 00:00:00.000 35 8

...

10454 2007-02-21 00:00:00.000 41 4

10453 2007-02-21 00:00:00.000 4 1

(25 row(s) affected)

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

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.

From a performance standpoint, you should evaluate indexing the ORDER BY columns to

support the TOP and OFFSET-FETCH options. Such indexing serves a very similar purpose to

indexing filtered columns and can help avoid scanning unnecessary data as well as sorting.

**Quick Check**

1. How do you guarantee deterministic results with TOP?

2. What are the benefits of using OFFSET-FETCH over TOP?

**Quick Check Answer**

1. By either returning all ties by using the WITH TIES option or by defining unique

ordering to break ties.

2. OFFSET-FETCH is standard and TOP isn’t; also, OFFSET-FETCH supports a skipping

capability that TOP doesn’t.

Practice **Filtering Data with TOP and OFFSET -FETCH**

In this practice, you exercise your knowledge of filtering data with TOP and OFFSET-FETCH.

If you encounter a problem completing an exercise, you can install the completed projects

from the Solution folder that is provided with the companion content for this chapter and

lesson.

Exercise 1 Use the TOP Option

In this exercise, you practice using the TOP option to filter data.

1. Open SSMS and connect to the sample database TSQL2012.

2. You are tasked with writing a query against the Production.Products table, returning

the five most expensive products from category 1. Write the following query.

SELECT TOP (5) productid, unitprice

FROM Production.Products

WHERE categoryid = 1

ORDER BY unitprice DESC;

You get the following result set.

productid unitprice

----------- ---------------------

38 263.50

43 46.00

2 19.00

1 18.00

35 18.00

This query returns the desired result, except it doesn’t have any handling of ties. In other

words, the ordering among products with the same unit price is nondeterministic.

3. You are requested to provide solutions to turn the previous query into a deterministic

one—one solution that includes ties and another that breaks the ties. First, address

the version that includes all ties by using the WITH TIES option. Add this option to the

query, as follows.

SELECT TOP (5) WITH TIES productid, unitprice

FROM Production.Products

WHERE categoryid = 1

ORDER BY unitprice DESC;

You get the following output, which includes ties.

productid unitprice

----------- ---------------------

38 263.50

43 46.00

2 19.00

1 18.00

39 18.00

35 18.00

76 18.00

4. Address the second version that breaks the ties by using productid, descending, as

follows.

SELECT TOP (5) productid, unitprice

FROM Production.Products

WHERE categoryid = 1

ORDER BY unitprice DESC, productid DESC;

This query generates the following output.

productid unitprice

----------- ---------------------

38 263.50

43 46.00

2 19.00

76 18.00

39 18.00

Exercise 2 Use the OFFSET-FETCH Option

In this exercise, you practice using the OFFSET-FETCH option to filter data.

1. Open SSMS and connect to the sample database TSQL2012.

2. You are requested to write a set of queries that page through products, five at a time,

in unit price ordering, using the product ID as the tie breaker. Start by writing a query

that returns the first five products.

SELECT productid, categoryid, unitprice

FROM Production.Products

ORDER BY unitprice, productid

OFFSET 0 ROWS FETCH FIRST 5 ROWS ONLY;

You could have used either the FIRST or the NEXT keyword, but say you decided to use

FIRST because it was the more natural option when not skipping any rows. This query

generates the following output.

productid categoryid unitprice

----------- ----------- ---------------------

33 4 2.50

24 1 4.50

13 8 6.00

52 5 7.00

54 6 7.45

3. Next, write a query that returns the next five rows (rows 6 through 10) using the following

query.

SELECT productid, categoryid, unitprice

FROM Production.Products

ORDER BY unitprice, productid

OFFSET 5 ROWS FETCH NEXT 5 ROWS ONLY;

This time, use the NEXT keyword because you are skipping some rows. This query

generates the following output.

productid categoryid unitprice

----------- ----------- ---------------------

75 1 7.75

23 5 9.00

19 3 9.20

45 8 9.50

47 3 9.50

4. Similarly, write the following query to return rows 11 through 15:

SELECT productid, categoryid, unitprice

FROM Production.Products

ORDER BY unitprice, productid

OFFSET 10 ROWS FETCH NEXT 5 ROWS ONLY;

This query generates the following output.

productid categoryid unitprice

----------- ----------- ---------------------

41 8 9.65

3 2 10.00

21 3 10.00

74 7 10.00

46 8 12.00

You would follow a similar process for subsequent pages.

Lesson Summary

■■ With the TOP and OFFSET-FETCH options, you can filter data based on a specified

number of rows and ordering.

■■ The ORDER BY clause that is normally used in the query for presentation ordering is

also used by TOP and OFFSET FETCH to indicate which rows to filter.

■■ The TOP option is a proprietary T-SQL feature that you can use to indicate a number or

a percent of rows to filter.

■■ You can make a TOP query deterministic in two ways: one is by using the WITH TIES

option to return all ties, and the other is by using unique ordering to break ties.

■■ The OFFSET-FETCH option is a standard option similar to TOP, supported by SQL Server

2012. Unlike TOP, it allows you to specify how many rows to skip before indicating how

many rows to filter. As such, it can be used for ad-hoc paging purposes.

■■ Both TOP and OFFSET-FETCH support expressions as inputs and not just constants.

Lesson Review

Answer the following questions to test your knowledge of the information in this lesson. You

can find the answers to these questions and explanations of why each answer choice is correct

or incorrect in the “Answers” section at the end of this chapter.

1. You execute a query with a TOP (3) option. Which of the following options most accurately

describes how many rows will be returned?

A. Fewer than three rows

B. Three rows or fewer

C. Three rows

D. Three rows or more

E. More than three rows

F. Fewer than three, three, or more than three rows

2. You execute a query with TOP (3) WITH TIES and nonunique ordering. Which of the

following options most accurately describes how many rows will be returned?

A. Fewer than three rows

B. Three rows or fewer

C. Three rows

D. Three rows or more

E. More than three rows

F. Fewer than three, three, or more than three rows

3. Which of the following OFFSET-FETCH options are valid in T-SQL? (Choose all that apply.)

A. SELECT … ORDER BY orderid OFFSET 25 ROWS

B. SELECT … ORDER BY orderid FETCH NEXT 25 ROWS ONLY

C. SELECT … ORDER BY orderid OFFSET 25 ROWS FETCH NEXT 25 ROWS ONLY

D. SELECT … <no ORDER BY> OFFSET 0 ROWS FETCH FIRST 25 ROWS ONLY

**Case Scenarios**

In the following case scenarios, you apply what you’ve learned about filtering and sorting data.

You can find the answers to these questions in the “Answers” section at the end of this chapter.

Case Scenario 1: Filtering and Sorting Performance

Recommendations

You are hired as a consultant to help address query performance problems in a beer factory

running SQL Server 2012. You trace a typical workload submitted to the system and observe

very slow query run times. You see a lot of network traffic. You see that many queries return

all rows to the client and then the client handles the filtering. Queries that do filter data often

manipulate the filtered columns. All queries have ORDER BY clauses, and when you inquire

about this, you are told that it’s not really needed, but the developers got accustomed to doing

so—just in case. You identify a lot of expensive sort operations. The customer is looking for

recommendations to improve performance and asks you the following questions:

1. Can anything be done to improve the way filtering is handled?

2. Is there any harm in specifying ORDER BY even when the data doesn’t need to be

returned ordered?

3. Any recommendations related to queries with TOP and OFFSET-FETCH?

Case Scenario 2: Tutoring a Junior Developer

You are tutoring a junior developer regarding filtering and sorting data with T-SQL. The developer

seems to be confused about certain topics and poses some questions to you. Answer

the following to the best of your knowledge:

1. When I try to refer to a column alias that I defined in the SELECT list in the WHERE

clause, I get an error. Can you explain why this isn’t allowed and what the workarounds

are?

2. Referring to a column alias in the ORDER BY clause seems to be supported. Why is

that?

3. Why is it that Microsoft made it mandatory to specify an ORDER BY clause when using

OFFSET-FETCH but not when using TOP? Does this mean that only TOP queries can

have nondeterministic ordering?

**Suggested Practices**

To help you successfully master the exam objectives presented in this chapter, complete the

following tasks.

Identify Logical Query Processing Phases and

Compare Filters

To practice your knowledge of logical query processing, list the elements you’ve learned

about so far in their right order.

■■ **Practice 1** In this chapter, you learned about using the WHERE clause to filter data

based on predicates, the ORDER BY clause to sort data, and the TOP and OFFSETFETCH

options as another way to filter data. Combined with your knowledge from

Chapter 1, list the query elements SELECT, FROM, WHERE, GROUP BY, HAVING, ORDER

BY, TOP, and OFFSET-FETCH in correct logical query processing order. Note that

because TOP and OFFSET-FETCH cannot be combined in the same query, you need to

create two such lists.

■■ **Practice 2** List the capabilities that the OFFSET-FETCH filter has that aren’t available

to TOP in SQL Server 2012, and also the other way around.

Understand Determinism

Recall that a deterministic query is one that has only one correct result. To demonstrate your

knowledge of query determinism, provide examples for deterministic and nondeterministic

queries.

■■ **Practice 1** Provide examples for queries with deterministic and nondeterministic

ordering. Describe in your own words what is required to get deterministic ordering.

■■ **Practice 2** Provide examples for deterministic and nondeterministic queries by using

TOP and OFFSET-FETCH. Explain how you can enforce determinism in both cases.

**Answers**

This section contains the answers to the lesson review questions and solutions to the case

scenarios in this chapter.

Lesson 1

1. **Correct Answer: B**

A. **Incorrect:** NULL is not part of the three possible logical results of a predicate in

T-SQL.

B. **Correct:** Three-valued logic refers to true, false, and unknown.

C. **Incorrect:** 1, 0, and NULL are not part of the three possible logical results of a

predicate.

D. **Incorrect:** -1, 0, and 1 are not part of the three possible logical results of a

predicate.

2. **Correct Answers: A, B, and C**

A. **Correct:** The form '2012-02-12' is language-neutral for the data types DATE,

DATETIME2, and DATETIMEOFFSET, but language-dependent for DATETIME and

SMALLDATETIME.

B. **Correct:** The form '02/12/2012' is language-dependent.

C. **Correct:** The form '12/02/2012' is language-dependent.

D. **Incorrect:** The form '20120212' is language-neutral.

3. **Correct Answers: B and E**

A. **Incorrect:** This predicate applies manipulation to the filtered column, and hence

isn’t a search argument.

B. **Correct:** The LIKE predicate is a search argument when the pattern starts with a

known prefix.

C. **Incorrect:** The LIKE predicate isn’t a search argument when the pattern starts with

a wild card.

D. **Incorrect:** The LIKE predicate isn’t a search argument when the pattern starts with

a wild card.

E. **Correct:** Because no manipulation is applied to the filtered column, the predicate

is a search argument.

Lesson 2

1. **Correct Answer: A**

A. **Correct:** Without an ORDER BY clause, ordering isn’t guaranteed and is said to be

arbitrary—it’s optimization-dependent.

B. **Incorrect:** Without an ORDER BY clause, there’s no guarantee for ordering.

C. **Incorrect:** Without an ORDER BY clause, there’s no guarantee for ordering.

D. **Incorrect:** Without an ORDER BY clause, there’s no guarantee for ordering.

2. **Correct Answer: C**

A. **Incorrect:** This uses ascending ordering for orderdate and descending just for

orderid.

B. **Incorrect:** This is invalid syntax.

C. **Correct:** The correct syntax is to specify DESC after each expression whose ordering

direction needs to be descending.

D. **Incorrect:** This is invalid syntax.

3. **Correct Answer: B, C, and D**

A. **Incorrect:** This is invalid syntax.

B. **Correct:** The default direction is ascending, so this clause uses ascending order for

both orderdate and orderid.

C. **Correct:** This clause explicitly uses ascending order for both orderdate and orderid.

D. **Correct:** The default direction is ascending, so this clause uses ascending order for

both orderdate and orderid.

Lesson 3

1. **Correct Answer: B**

A. **Incorrect:** If there are at least three rows in the query result without TOP, the

query will return three rows.

B. **Correct:** If there are fewer rows than three in the query result without TOP, the

query will return only those rows. If there are three rows or more without TOP, the

query will return three rows.

C. **Incorrect:** If there are fewer rows than three in the query result without TOP, the

query will return only those rows.

D. **Incorrect:** Unless the WITH TIES option is used, the query won’t return more than

the requested number of rows.

E. **Incorrect:** Unless the WITH TIES option is used, the query won’t return more than

the requested number of rows.

F. **Incorrect:** Unless the WITH TIES option is used, the query won’t return more than

the requested number of rows.

2. **Correct Answer: F**

A. **Incorrect:** If there are at least three rows in the query result without TOP, the

query will return at least three rows.

B. **Incorrect:** If there are more than three rows in the result, as well as ties with the

third row, the query will return more than three rows.

C. **Incorrect:** If there are fewer rows than three in the query result without TOP, the

query will return only those rows. If there are more than three rows in the result, as

well as ties with the third row, the query will return more than three rows.

D. **Incorrect:** If there are fewer rows than three in the query result without TOP, the

query will return only those rows.

E. **Incorrect:** If there are three rows or less in the query result without TOP, the query

won’t return more than three rows.

F. **Correct:** If there are fewer rows than three in the query result without TOP, the

query will return only those rows. If there are at least three rows in the result and

no ties with the third, the query will return three rows. If there are more than three

rows in the result, as well as ties with the third row, the query will return more than

three rows.

3. **Correct Answer: A and C**

A. **Correct:** T-SQL supports indicating an OFFSET clause without a FETCH clause.

B. **Incorrect:** Contrary to standard SQL, T-SQL does not support a FETCH clause without

an OFFSET clause.

C. **Correct:** T-SQL supports indicating both OFFSET and FETCH clauses.

D. **Incorrect:** T-SQL does not support OFFSET-FETCH without an ORDER BY clause.

Case Scenario 1

1. For one thing, as much filtering as possible should be done in the database. Doing most

of the filtering in the client means that you’re scanning more data, which increases the

stress on the storage subsystem, and also that you cause unnecessary network traffic.

When you do filter in the databases, for example by using the WHERE clause, you

should use search arguments that increase the likelihood for efficient use of indexes.

You should try as much as possible to avoid manipulating the filtered columns.

2. Adding an ORDER BY clause means that SQL Server needs to guarantee returning the

rows in the requested order. If there are no existing indexes to support the ordering

requirements, SQL Server will have no choice but to sort the data. Sorting is expensive

with large sets. So the general recommendation is to avoid adding ORDER BY clauses

to queries when there are no ordering requirements. And when you do need to return

the rows in a particular order, consider arranging supporting indexes that can prevent

SQL Server from needing to perform expensive sort operations.

3. The main way to help queries with TOP and OFFSET-FETCH perform well is by arranging

indexes to support the ordering elements. This can prevent scanning all data, in

addition to sorting.

Case Scenario 2

1. To be able to understand why you can’t refer to an alias that was defined in the

SELECT list in the WHERE clause, you need to understand logical query processing.

Even though the keyed-in order of the clauses is SELECT-FROM-WHERE-GROUP

BY-HAVING-ORDER BY, the logical query processing order is FROM-WHERE-GROUP

BY-HAVING-SELECT-ORDER BY. As you can see, the WHERE clause is evaluated prior to

the SELECT clause, and therefore aliases defined in the SELECT clause aren’t visible to

the WHERE clause.

2. Logical query processing order explains why the ORDER BY clause can refer to aliases

defined in the SELECT clause. That’s because the ORDER BY clause is logically evaluated

after the SELECT clause.

3. The ORDER BY clause is mandatory when using OFFSET-FETCH because this clause is

standard, and standard SQL decided to make it mandatory. Microsoft simply followed

the standard. As for TOP, this feature is proprietary, and when Microsoft designed it,

they chose to allow using TOP in a completely nondeterministic manner—without an

ORDER BY clause. Note that the fact that OFFSET-FETCH requires an ORDER BY clause

doesn’t mean that you must use deterministic ordering. For example, if your ORDER

BY list isn’t unique, the ordering isn’t deterministic. And if you want the ordering to be

completely nondeterministic, you can specify ORDER BY (SELECT NULL) and then it’s

equivalent to not specifying an ORDER BY clause at all