Module 5: Sorting and Filtering Data

Contents:

Module Overview

In this module, you will learn how to enhance a query to limit the number of rows that the query returns, and control the order in which the rows are displayed.

Earlier in this course, you learned that, according to relational theory, sets of data do not include any definition of a sort order. Therefore, if you require the output of a query to be displayed in a certain order, you should add an ORDER BY clause to your SELECT statement. In this module, you will learn how to write a query using ORDER BY to control the display order.

You have already learned how to build a FROM clause to return rows from one or more tables. It is unlikely that you will always want to return all rows from the source. For performance reasons, in addition to the needs of your client application or report, you will want to limit which rows are returned. As you will learn in this module, you can limit the rows selected with a WHERE clause based on a predicate; you can also limit the number of rows with a TOP, or OFFSET and FETCH clause, based on the order of the rows selected.

When you work with real-world data in queries, you may encounter situations where values are missing. It is important to write queries that can handle missing values correctly. In this module, you will learn about handling missing and unknown results.

Objectives

Filter data with predicates in the WHERE clause:

- Sort data using ORDER BY.
- Filter data in the SELECT clause with TOP.
- Filter data with OFFSET and FETCH.

Lesson 1: Sorting Data

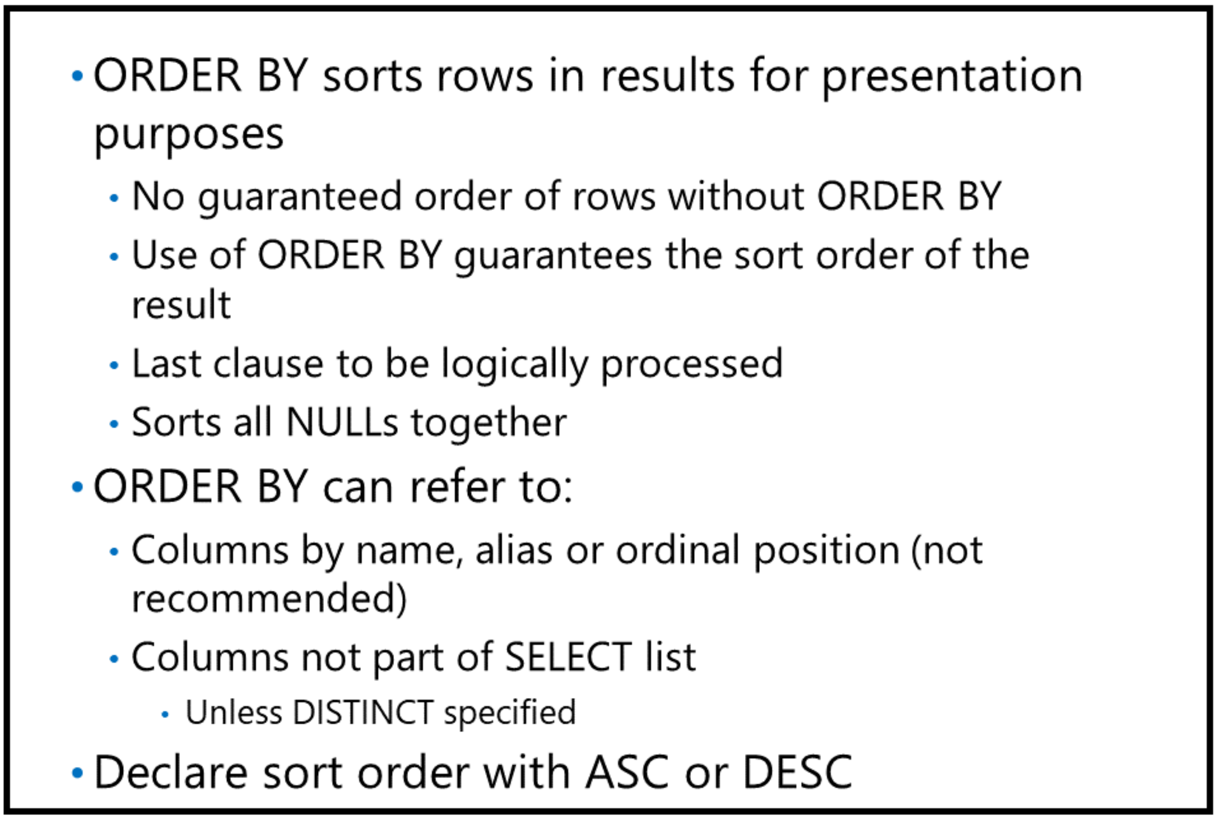
In this lesson, you will learn how to add an ORDER BY clause to a query to control the order of rows displayed in the output of the query.

Lesson Objectives

After completing this lesson, you will be able to:

- Describe the ORDER BY clause.
- Describe the ORDER BY clause syntax.
- List examples of the ORDER BY clause.

Using the ORDER BY Clause



- ORDER BY sorts rows in results for presentation purposes
  - No guaranteed order of rows without ORDER BY
  - Use of ORDER BY guarantees the sort order of the result
  - Last clause to be logically processed
  - Sorts all NULLs together
- ORDER BY can refer to:
  - Columns by name, alias or ordinal position (not recommended)
  - Columns not part of SELECT list
    - Unless DISTINCT specified
- Declare sort order with ASC or DESC

In the logical order of query processing, ORDER BY is the last phase of a SELECT statement to be executed. ORDER BY enables you to control the sorting of rows as they are output from the query to the client application. Without an ORDER BY clause, SQL Server does not guarantee the order of rows—in keeping with relational theory.

ORDER BY Clause

```
SELECT   <select_list>
FROM     <table_source>
ORDER BY <order_by_list> [ASC|DESC];
```

ORDER BY can take several types of element in its list:

Columns by name. Additional columns beyond the first specified in the list will be used as tiebreakers for nonunique values in the first column.

Column aliases. Remember that ORDER BY is processed after the SELECT clause and therefore has access to aliases defined in the SELECT list.

Columns by position in the SELECT clause. This is not recommended, because of diminished readability and the extra care required to keep the ORDER BY list up to date with any changes made to the SELECT list column order.

- **Columns not detailed in the SELECT list, but part of tables listed in the FROM clause**. If the query uses a DISTINCT option, any columns in the ORDER BY list must be included in the SELECT list.

Note: ORDER BY may also include a COLLATE clause, which provides a way to sort by a specific character collation, instead of the collation of the column in the table. Collations will be discussed further later in this course.

In addition to specifying which columns should be used to determine the sort order, you may also control the direction of the sort by using ASC for ascending (A-Z, 0-9) or DESC for descending (Z-A, 9-0). Ascending sorts are the default. Each column may be provided with a separate order, as in the following example:

Ascending and Descending Sort

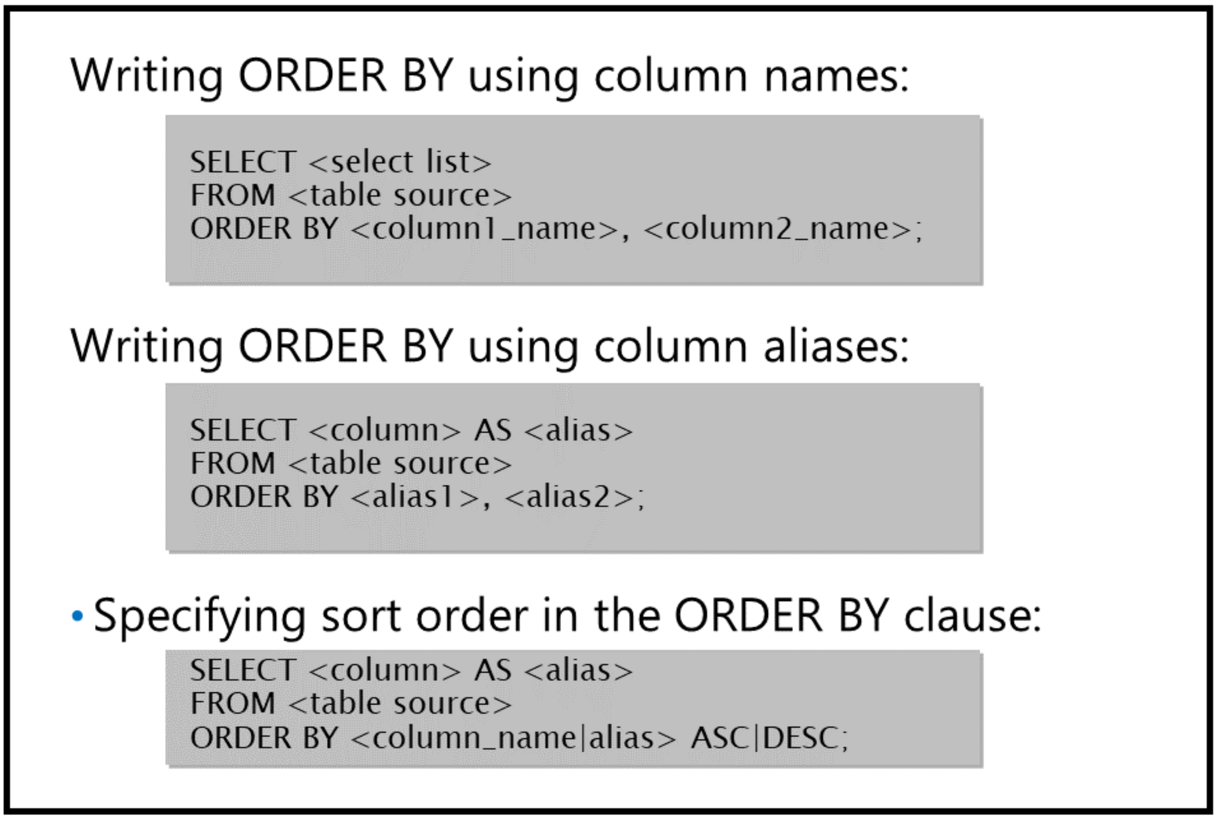
```
USE TSQL;
GO
SELECT hiredate, firstname, lastname
FROM HR.Employees
ORDER BY hiredate DESC, lastname ASC;
```

For additional information on the ORDER BY clause, see Microsoft Docs:

ORDER BY Clause (Transact-SQL)

http://go.microsoft.com/fwlink/?LinkID=402718

ORDER BY Clause Syntax



Writing ORDER BY using column names:

```
SELECT <select list>
FROM <table source>
ORDER BY <column1_name>, <column2_name>;
```

Writing ORDER BY using column aliases:

```
SELECT <column> AS <alias>
FROM <table source>
ORDER BY <alias1>, <alias2>;
```

- Specifying sort order in the ORDER BY clause:

```
SELECT <column> AS <alias>
FROM <table source>
ORDER BY <column_name|alias> ASC|DESC;
```

ORDER BY Clause

```
ORDER BY <order_by_list>
OFFSET <offset_value> ROW|ROWS FETCH FIRST|NEXT <fetch_value>

ROW|ROWS ONLY
```

Note: The use of the OFFSET-FETCH option in the ORDER BY clause will be covered later in this module.

ORDER BY List

```
ORDER BY <column_name_1>, <column_name_2>;
```

ORDER BY List Example

```
ORDER BY country, region, city;
```

ORDER BY an Alias

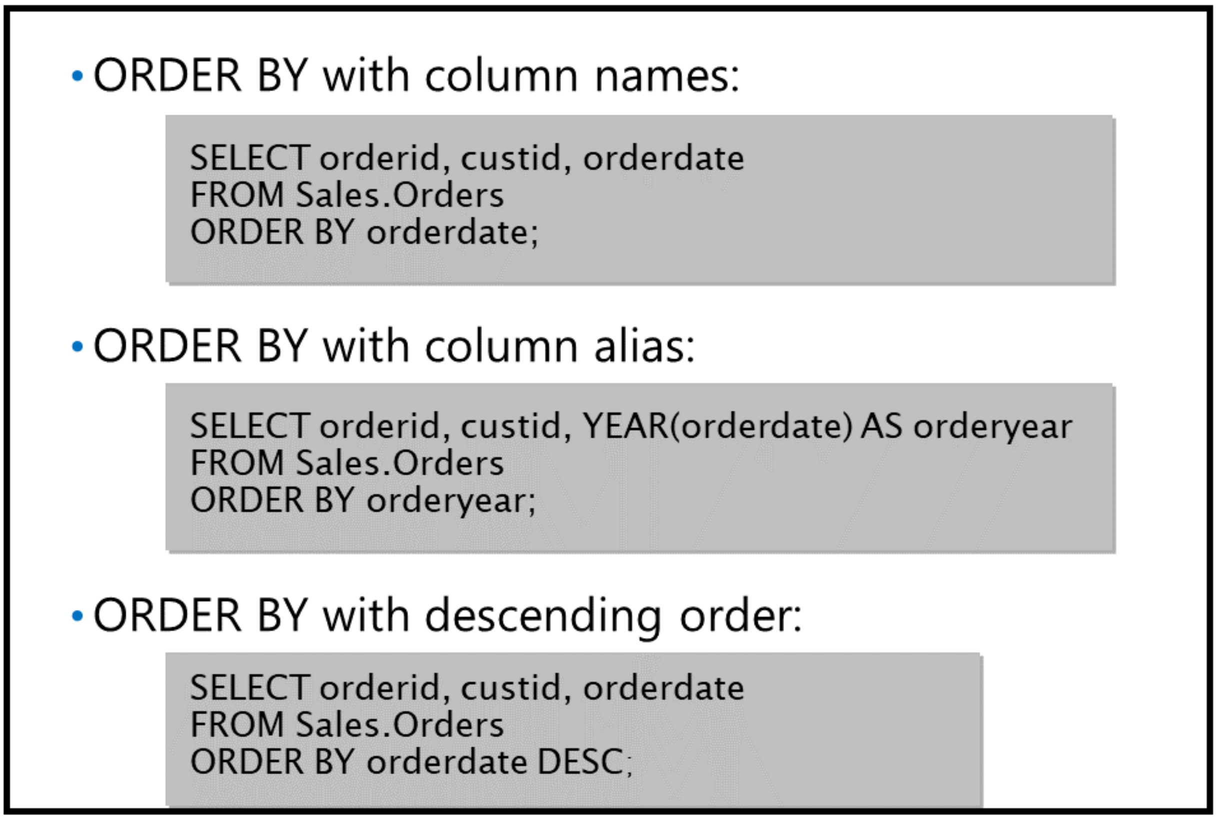
```
SELECT <column_name_1> AS alias1, <column_name_2> AS alias2
FROM <table source>
ORDER BY alias1;
```

ORDER BY Using Column Alias Example

```
SELECT orderid, custid, YEAR(orderdate) AS orderyear
FROM Sales.Orders
ORDER BY orderyear;
```

Note: See the previous topic for the syntax and usage of ASC or DESC to control sort order.

ORDER BY Clause Examples



- ORDER BY with column names:

```
SELECT orderid, custid, orderdate
FROM Sales.Orders
ORDER BY orderdate;
```

- ORDER BY with column alias:

```
SELECT orderid, custid, YEAR(orderdate) AS orderyear
FROM Sales.Orders
ORDER BY orderyear;
```

- ORDER BY with descending order:

```
SELECT orderid, custid, orderdate
FROM Sales.Orders
ORDER BY orderdate DESC;
```

The following are examples of common queries using ORDER BY to sort the output for display. All queries use the TSQL sample database.

ORDER BY Example 1

```
SELECT orderid, custid, orderdate
FROM Sales.Orders
ORDER BY orderdate;
```

ORDER BY Example 2

```
SELECT orderid, custid, orderdate
FROM Sales.Orders
ORDER BY orderdate DESC;
```

ORDER BY Example 3

```
SELECT hiredate, firstname, lastname
FROM HR.Employees
ORDER BY hiredate DESC, lastname ASC;
```

Demonstration: Sorting Data

In this demonstration, you will see how to sort data using the ORDER BY clause.

Demonstration Steps

Sort Data Using The ORDER BY Clause

1. Ensure that the **MT17B-WS2016-NAT**, **20761C-MIA-DC**, and **20761C-MIA-SQL** virtual machines are running, and then log on to **20761C-MIA-SQL** as **ADVENTUREWORKS\Student** with the password **Pa55w.rd**.

2. Start SQL Server Management Studio and connect to your Azure instance of the **AdventureWorksLT** database engine instance using SQL Server authentication.
3. Open the **Demo.ssmssln** solution in the **D:\Demofiles\Mod05\Demo** folder.
4. In Solution Explorer, expand **Queries**, and then double-click **11 - Demonstration A.sql**.
5. In the **Available Databases** list, click **ADVENTUREWORKSLT**.
6. Select the code under the comment **Step 1**, and then click **Execute**.
7. Select the code under the comment **Step 2**, and then click **Execute**.
8. Select the code under the comment **Step 3**, and then click **Execute**.
9. Select the code under the comment **Step 4**, and then click **Execute**.
10. Select the code under the comment **Step 5**, and then click **Execute**.
11. Select the code under the comment **Step 6**, and then click **Execute**.
12. Keep SQL Server Management Studio open for the next demonstration.

Check Your Knowledge

Discovery

If you declare an alias for a column in the SELECT clause, you cannot use that alias in the WHERE clause—but you can use it in the ORDER BY clause. Why is this?

Show solution Reset

Lesson 2: Filtering Data with Predicates

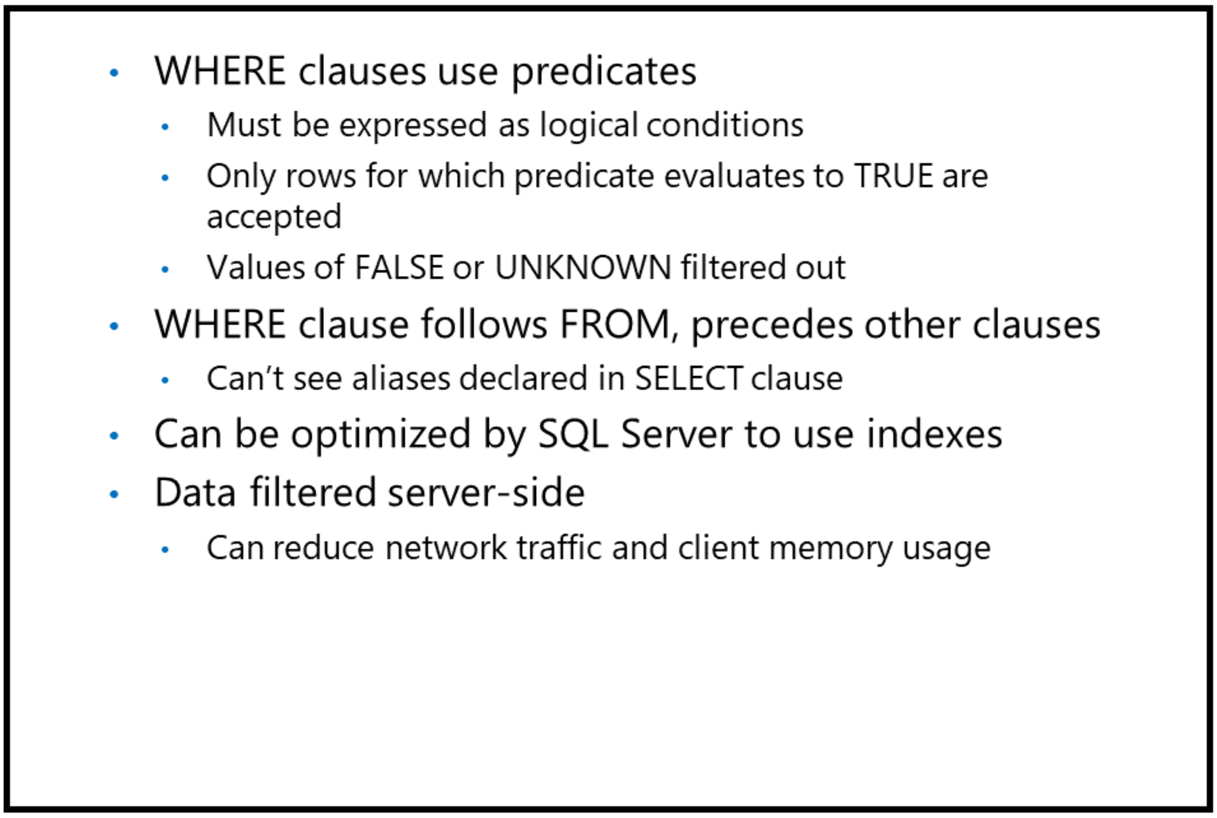
When querying SQL Server, you will mostly want to retrieve only a subset of all rows stored in the table(s) listed in the FROM clause. This is especially true as data volumes grow. To limit which rows are returned, you will typically use the WHERE clause in the SELECT statement. In this lesson, you will learn how to construct WHERE clauses to filter out rows that do not match the predicate.

Lesson Objectives

After completing this lesson, you will be able to:

- Describe the WHERE clause.
- Describe the syntax of the WHERE clause.

Filtering Data in the WHERE Clause with Predicates



- WHERE clauses use predicates
  - Must be expressed as logical conditions
  - Only rows for which predicate evaluates to TRUE are accepted
  - Values of FALSE or UNKNOWN filtered out
- WHERE clause follows FROM, precedes other clauses
  - Can't see aliases declared in SELECT clause
- Can be optimized by SQL Server to use indexes
- Data filtered server-side
  - Can reduce network traffic and client memory usage

To limit the rows that are returned by your query, you will need to add a WHERE clause to your SELECT statement, following the FROM clause. WHERE clauses are constructed from a search condition which, in turn, is written as a predicate expression. The predicate provides a logical filter through which each row must pass. Only rows returning TRUE in the predicate will be output to the next logical phase of the query.

When writing a WHERE clause, keep the following considerations in mind:

- Your predicate must be expressed as a logical condition, evaluating to TRUE or FALSE. (The evaluation may be NULL when working with missing values or NULL. See Lesson 4 for more information.)
- Only rows for which the predicate evaluates as TRUE will be passed through the filter.
- Values of FALSE or UNKNOWN will be filtered out.
- Column aliases declared in the SELECT clause of the query cannot be used in the WHERE clause predicate.
- Remember that, logically, the WHERE clause is the next phase in query execution after FROM, so the WHERE clause will be processed before other clauses, such as SELECT. One consequence of this is that the WHERE clause will be unable to refer to column aliases created in the SELECT clause. If you have created expressions in the SELECT list, you will need to repeat the expressions for use in the WHERE clause.

Filtering Example

```
SELECT orderid, custid, YEAR(orderdate) AS ordyear
FROM Sales.Orders
WHERE YEAR(orderdate) = 2006;
```

Incorrect Column Alias in WHERE Clause

```
SELECT orderid, custid, YEAR(orderdate) AS ordyear
FROM Sales.Orders
WHERE ordyear = 2006;
```

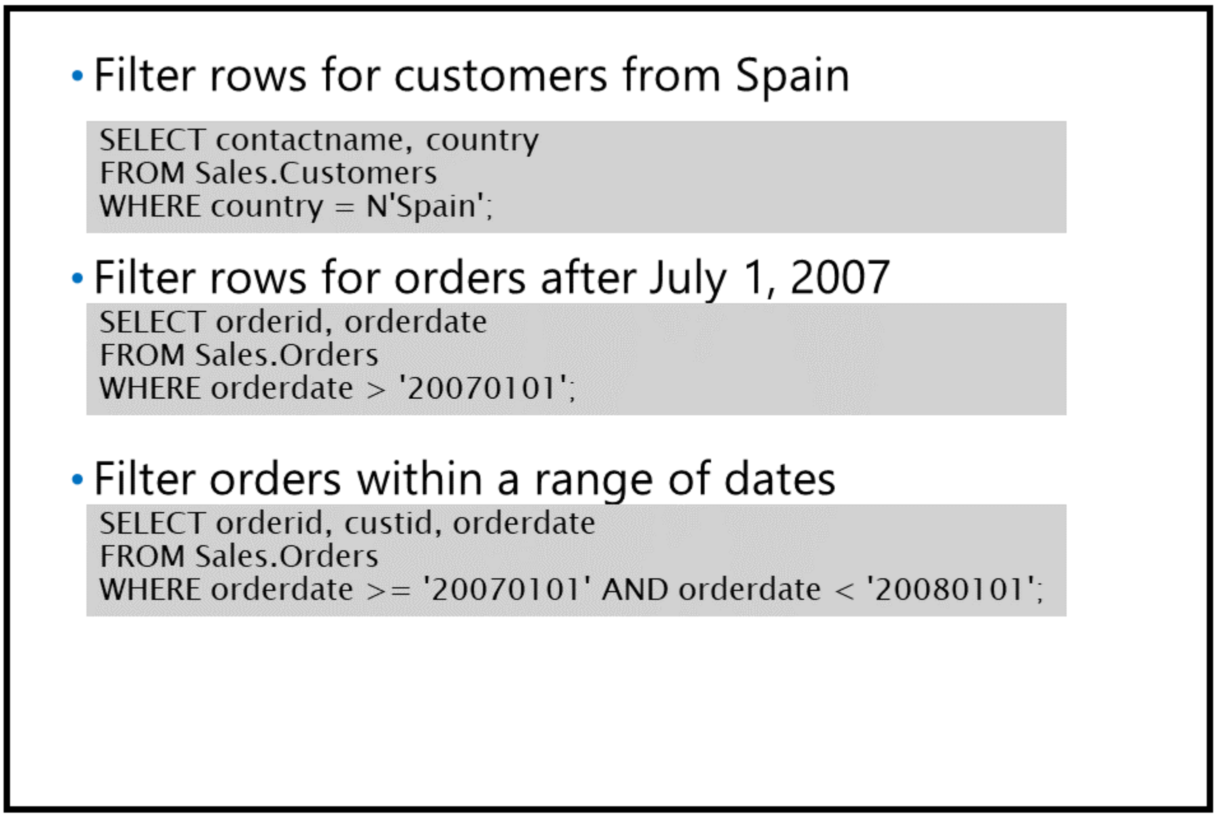
The error message points to the use of the column alias in Line 3 of the batch:

```
Msg 207, Level 16, State 1, Line 3
```

```
Invalid column name 'ordyear'.
```

From the perspective of query performance, the use of effective WHERE clauses can provide a significant positive impact on SQL Server. Rather than return all rows to the client for post-processing, a WHERE clause causes SQL Server to filter data on the server side. This can reduce network traffic and memory usage on the client. SQL Server developers and administrators can also create indexes to support commonly-used predicates, further improving performance.

WHERE Clause Syntax



- Filter rows for customers from Spain

```
SELECT contactname, country
FROM Sales.Customers
WHERE country = N'Spain';
```

- Filter rows for orders after July 1, 2007

```
SELECT orderid, orderdate
FROM Sales.Orders
WHERE orderdate > '20070101';
```

- Filter orders within a range of dates

```
SELECT orderid, custid, orderdate
FROM Sales.Orders
WHERE orderdate >= '20070101' AND orderdate < '20080101';
```

WHERE Clause Syntax

```
WHERE <search_condition>
```

Typical WHERE Clause

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

WHERE Clause Example

```
SELECT contactname, country
FROM Sales.Customers
WHERE country = N'Spain';
```

Any of the logical operators introduced in the T-SQL language module earlier in this course may be used in a WHERE clause predicate.

WHERE Clause Example

```
SELECT orderid, orderdate
FROM Sales.Orders
WHERE orderdate > '20070101';
```

Note: The representation of dates as strings delimited by quotation marks will be covered in the next module.

In addition to using logical operators, literals, or constants in a WHERE clause, you may also use several T-SQL options in your predicate:

Predicates and Operators	**Description**
IN	Determines whether a specified value matches any value in a subquery or a list.
BETWEEN	Specifies an inclusive range to test.
LIKE	Determines whether a specific character string matches a specified pattern.
AND	Combines two Boolean expressions and returns TRUE only when both are TRUE.
OR	Combines two Boolean expressions and returns TRUE if either is TRUE.

Predicates and Operators	Description
NOT	Reverses the result of a search condition.

Note: The use of LIKE to match patterns in character-based data will be covered in the next module.

WHERE with OR Example

```
SELECT custid, companyname, country
FROM Sales.Customers
WHERE country = N'UK' OR country = N'Spain';
```

WHERE with IN Example

```
SELECT custid, companyname, country
FROM Sales.Customers
WHERE country IN (N'UK',N'Spain');
```

Range Example

```
SELECT orderid, custid, orderdate
FROM Sales.Orders
WHERE orderdate >= '20070101' AND orderdate <= '20080630';
```

BETWEEN Operator

```
SELECT orderid, custid, orderdate
FROM Sales.Orders
WHERE orderdate BETWEEN '20070101' AND '20080630';
```

Note: The use of comparison operators with date and time data types requires special consideration. For more information, see Module 6.

Demonstration: Filtering Data with Predicates

In this demonstration, you will see how to filter data in a WHERE clause.

Demonstration Steps

Filter Data in a WHERE Clause

1. In Solution Explorer, open the **21 - Demonstration B.sql** script file.
2. In the **Available Databases** list, click **ADVENTUREWORKSLT**.
3. Select the code under the comment **Step 1**, and then click **Execute**.
4. Select the code under the comment **Step 2**, and then click **Execute**.
5. Select the code under the comment **Step 3**, and then click **Execute**. Note the error message.
6. Select the code under the comment **Step 4**, and then click **Execute**.
7. Select the code under the comment **Step 5**, and then click **Execute**.
8. Select the code under the comment **Step 6**, and then click **Execute**.
9. Select the code under the comment **Step 7**, and then click **Execute**.
10. Select the code under the comment **Step 8**, and then click **Execute**.
11. Select the code under the comment **Step 9**, and then click **Execute**.
12. Select the code under the comment **Step 10**, and then click **Execute**.
13. Select the code under the comment **Step 11**, and then click **Execute**.
14. Keep SQL Server Management Studio open for the next demonstration.

Check Your Knowledge

Discovery

You have a table named Employees that includes a column named StartDate. You want to find who started in any year other than 2014. What query would you use?

Show solution Reset

Lesson 3: Filtering Data with TOP and OFFSET-FETCH

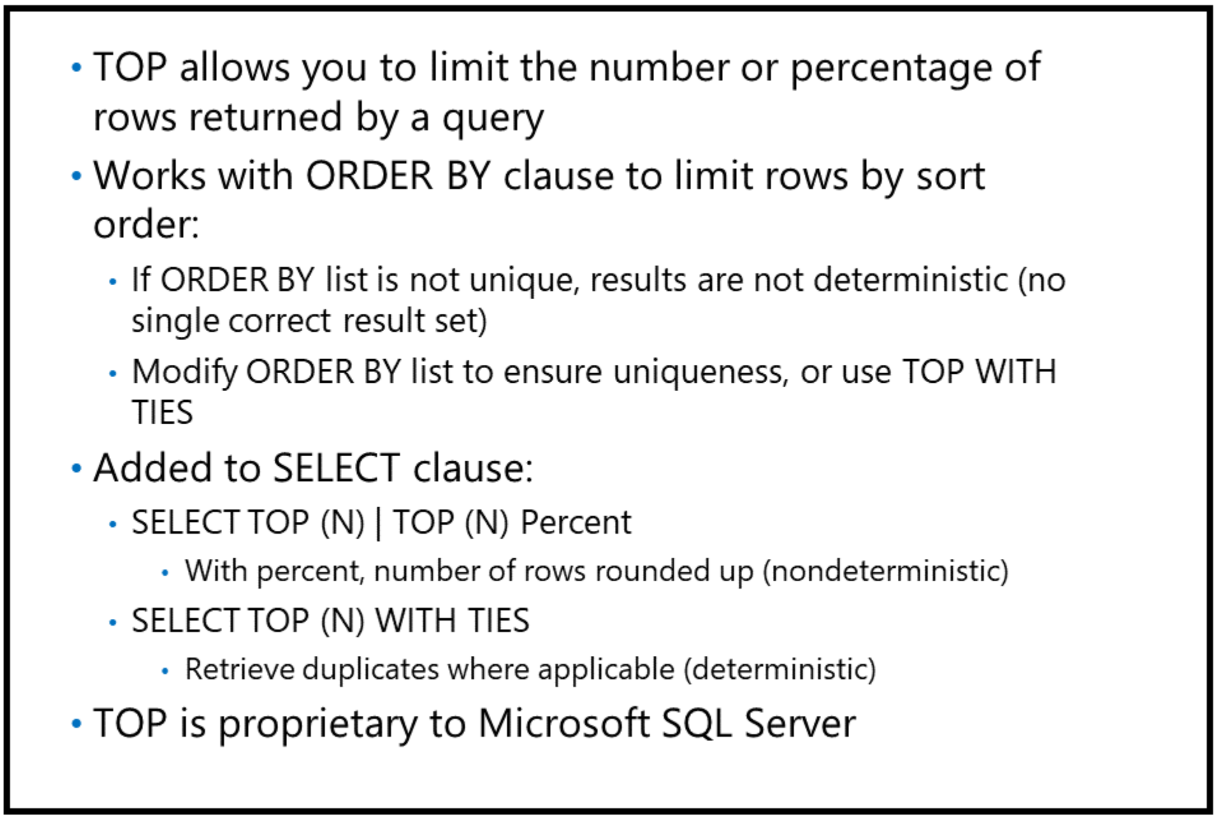
In the previous lesson, you wrote queries that filtered rows, based on data stored within them. You can also write queries that filter ranges of rows, based either on a specific number to retrieve, or one range of rows at a time. In this lesson, you will learn how to use a TOP option to limit ranges of rows in the SELECT clause. You will also learn how to limit ranges of rows using the OFFSET-FETCH option of an ORDER BY clause.

Lesson Objectives

After completing this lesson, you will be able to:

- Describe the TOP option.
- Describe the OFFSET-FETCH clause.
- Describe the syntax of the OFFSET-FETCH clause.

Filtering in the SELECT Clause Using the TOP Option



- TOP allows you to limit the number or percentage of rows returned by a query
- Works with ORDER BY clause to limit rows by sort order:
  - If ORDER BY list is not unique, results are not deterministic (no single correct result set)
  - Modify ORDER BY list to ensure uniqueness, or use TOP WITH TIES
- Added to SELECT clause:
  - SELECT TOP (N) | TOP (N) Percent
    - With percent, number of rows rounded up (nondeterministic)
  - SELECT TOP (N) WITH TIES
    - Retrieve duplicates where applicable (deterministic)
- TOP is proprietary to Microsoft SQL Server

When returning rows from a query, you may need to limit the total number of rows returned, in addition to filtering with a WHERE clause. The TOP option, a Microsoft-proprietary extension of the SELECT clause, will let you specify a number of rows to return, either as an ordinal number or as a percentage of all candidate rows.

TOP Option

```
SELECT TOP (N) <column_list>
FROM <table_source>
WHERE <search_condition>
ORDER BY <order list>;
```

TOP Example

```
SELECT TOP (5) orderid, custid, orderdate
FROM Sales.Orders
ORDER BY orderdate DESC;
```

Note: The TOP operator depends on an ORDER BY clause to provide meaningful precedence to the rows selected. In the absence of ORDER BY, there is no guarantee for which rows will be returned. In the previous example, any five orders might be returned if there wasn't an ORDER BY clause.
In addition to specifying a fixed number of rows to be returned, the TOP keyword also accepts the WITH TIES option, which will retrieve any rows with values that might be found in the selected top N rows.

Without the WITH TIES Option

```
SELECT TOP (5) orderid, custid, orderdate
FROM Sales.Orders
ORDER BY orderdate DESC;
```

The results show five rows with two distinct orderdate values:

```
orderid     custid      orderdate
----------- ----------- -----------------------

11077       65          2008-05-06 00:00:00.000
11076       9           2008-05-06 00:00:00.000
11075       68          2008-05-06 00:00:00.000
11074       73          2008-05-06 00:00:00.000
11073       58          2008-05-05 00:00:00.000
(5 row(s) affected)
```

With the WITH TIES Option

```
SELECT TOP (5) WITH TIES orderid, custid, orderdate
FROM Sales.Orders
ORDER BY orderdate DESC;
```

This modified query returns the following results:

```
orderid     custid      orderdate
----------- ----------- -----------------------
11077       65          2008-05-06 00:00:00.000
11076       9           2008-05-06 00:00:00.000
11075       68          2008-05-06 00:00:00.000
11074       73          2008-05-06 00:00:00.000
11073       58          2008-05-05 00:00:00.000
11072       20          2008-05-05 00:00:00.000
11071       46          2008-05-05 00:00:00.000
11070       44          2008-05-05 00:00:00.000

(8 row(s) affected)
```

The decision to include WITH TIES will depend on your knowledge of the source data, its potential for unique values, and the requirements of the query you are writing.

To return a percentage of the row count, use the PERCENT option with TOP instead of a fixed number.

Returning a Percentage of Records

```
SELECT TOP (10) PERCENT orderid, custid, orderdate
FROM Sales.Orders
ORDER BY orderdate DESC;
```

TOP (N) PERCENT may also be used with the WITH TIES option.

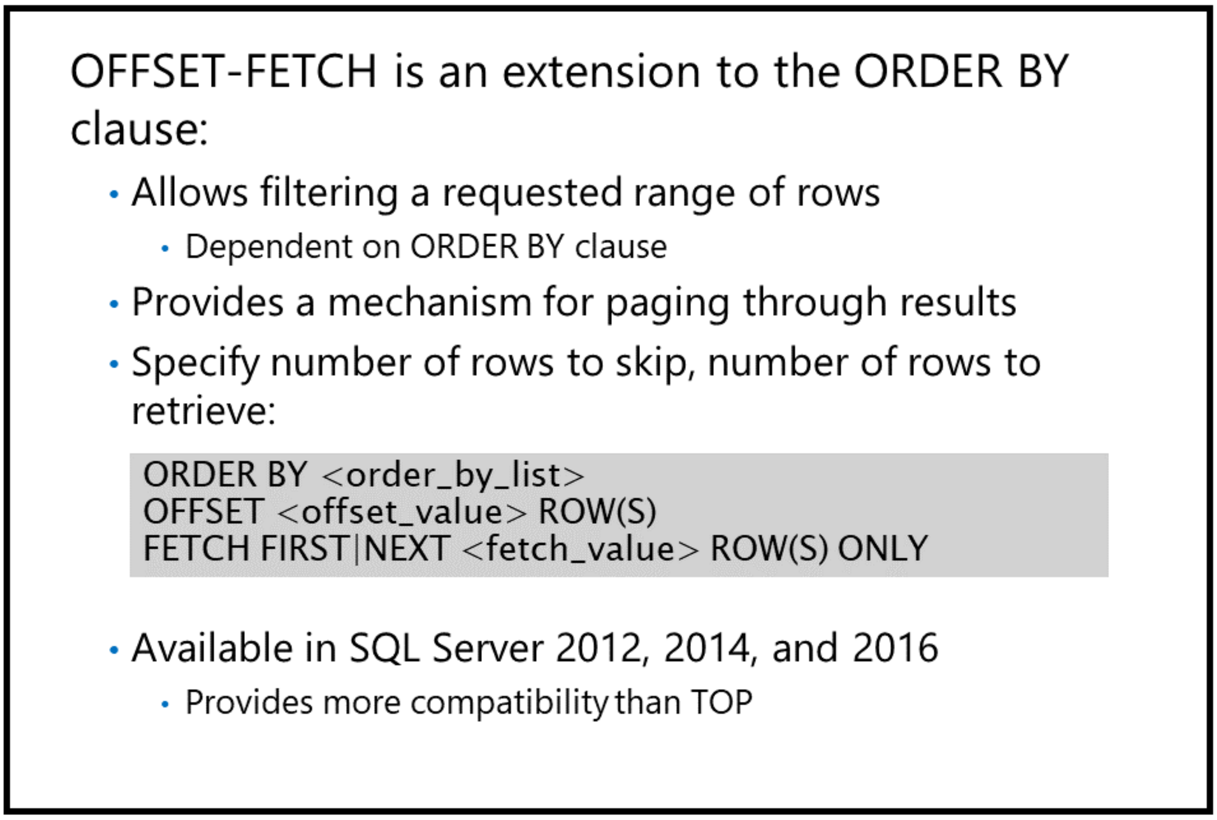
> **Note:** For the purposes of row count, TOP (N) PERCENT will round up to the nearest integer.

For additional information about the TOP clause, see Microsoft Docs:

TOP (Transact-SQL)

http://go.microsoft.com/fwlink/?LinkID=402719

Filtering in the ORDER BY Clause Using OFFSET-FETCH



OFFSET-FETCH is an extension to the ORDER BY clause:

- Allows filtering a requested range of rows
  - Dependent on ORDER BY clause
- Provides a mechanism for paging through results
- Specify number of rows to skip, number of rows to retrieve:

```
ORDER BY <order_by_list>
OFFSET <offset_value> ROW(S)
FETCH FIRST|NEXT <fetch_value> ROW(S) ONLY
```

- Available in SQL Server 2012, 2014, and 2016
  - Provides more compatibility than TOP

While the TOP option is used by many SQL Server professionals as a method for retrieving only a certain range of rows, it also has disadvantages:

- TOP is proprietary to T-SQL and SQL Server.
- TOP does not support skipping a range of rows.
- Because TOP depends on an ORDER BY clause, you cannot use one sort order to establish the rows filtered by TOP and another to determine the output display.

To address a number of these concerns, Microsoft added the OFFSET-FETCH extension to the ORDER BY clause.

Like TOP, OFFSET-FETCH enables you to return only a range of the rows selected by your query. However, it adds the functionality to supply a starting point (an offset) and a value to specify how many rows you would like to return (a fetch value). This provides a convenient technique for paging through results.

When paging, you will need to consider that each query with an OFFSET-FETCH clause runs independently of any previous or subsequent query. There is no server-side state maintained, and you will need to track your position through a result set via client-side code.

As you will see in the next topic, OFFSET-FETCH has been written to allow a more natural English language syntax.

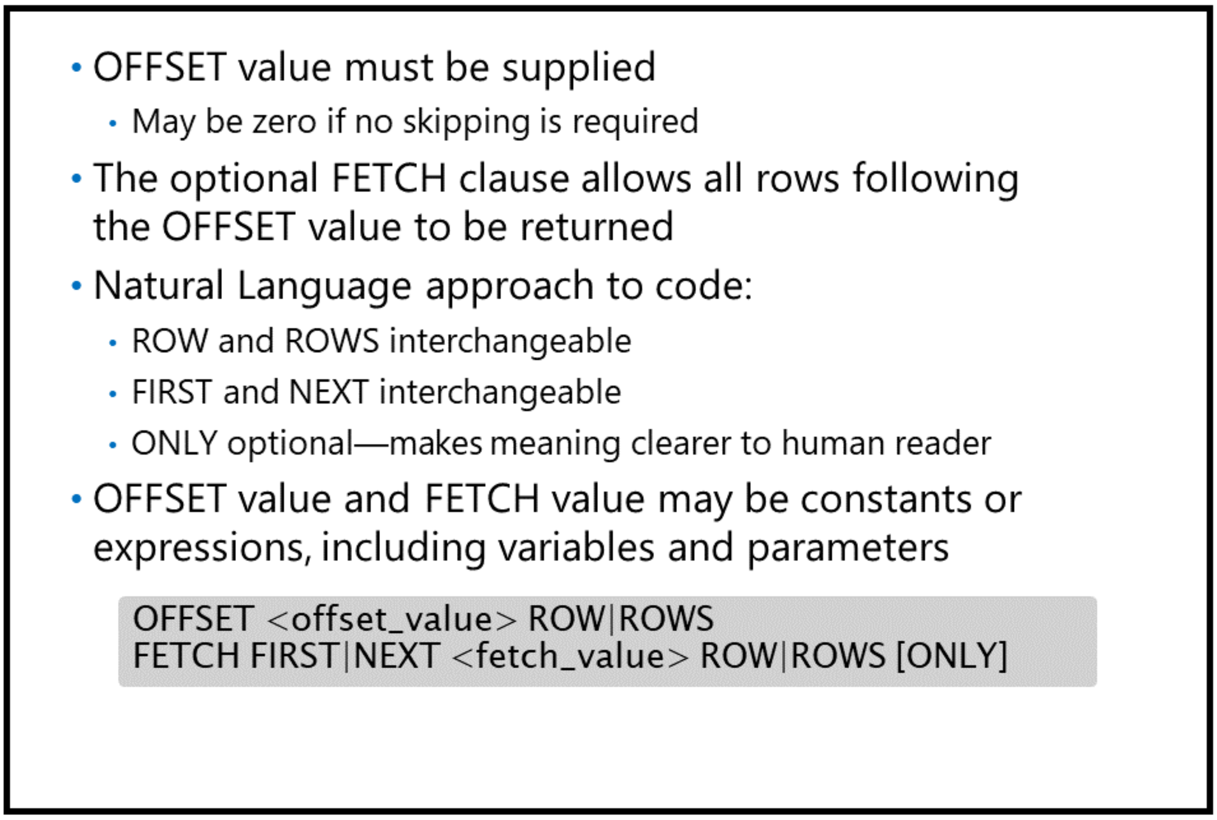
OFFSET-FETCH is supported in SQL Server 2012, 2014, and 2016.

For more information about the OFFSET-FETCH clause, see *Using OFFSET and FETCH to limit the rows returned* in Microsoft Docs:

ORDER BY Clause (Transact-SQL)

http://go.microsoft.com/fwlink/?LinkID=402718

OFFSET-FETCH Syntax



- OFFSET value must be supplied
  - May be zero if no skipping is required
- The optional FETCH clause allows all rows following the OFFSET value to be returned
- Natural Language approach to code:
  - ROW and ROWS interchangeable
  - FIRST and NEXT interchangeable
  - ONLY optional—makes meaning clearer to human reader
- OFFSET value and FETCH value may be constants or expressions, including variables and parameters

```
OFFSET <offset_value> ROW|ROWS
FETCH FIRST|NEXT <fetch_value> ROW|ROWS [ONLY]
```

OFFSET-FETCH Clause

```
OFFSET { integer_constant | offset_row_count_expression } { ROW |
ROWS }

    [FETCH { FIRST | NEXT } {integer_constant |
fetch_row_count_expression } { ROW | ROWS } ONLY]
```

To use OFFSET-FETCH, you will supply a starting OFFSET value (which may be zero) and an optional number of rows to return, as in the following example:

OFFSET FETCH Example 1

```
SELECT orderid, custid, orderdate
FROM Sales.Orders
ORDER BY orderdate, orderid DESC
OFFSET 10 ROWS FETCH NEXT 10 ROWS ONLY;
```

As you can see in the syntax definition, the OFFSET clause is required, but the FETCH clause is not. If the FETCH clause is omitted, all rows following OFFSET will be returned. You will also find that the keywords ROW and ROWS are interchangeable, as are FIRST and NEXT, which enables a more natural syntax.

To ensure the accuracy of the results, especially as you move from page to page of data, it is important to construct an ORDER BY clause that will provide unique ordering and yield a deterministic result. Although unlikely, due to SQL Server's query optimizer, it is technically possible for a row to appear on more than one page, unless the range of rows is deterministic.

Note: To use OFFSET-FETCH for paging, you might supply the OFFSET value, in addition to row count expressions, in the form of variables or parameters. You will learn more about variables and stored procedure parameters in later modules of this course.

The following are some examples of using OFFSET-FETCH in T-SQL queries—all of them use the AdventureWorks sample database:

OFFSET-FETCH Example 2

```
SELECT orderid, custid, empid, orderdate
FROM Sales.Orders
ORDER BY orderdate DESC
OFFSET 0 ROWS FETCH FIRST 50 ROWS ONLY;
```

OFFSET-FETCH Example 3

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

Note: Unlike those found in any previous modules, examples of OFFSET-FETCH must be executed by SQL Server 2012 or later. OFFSET-FETCH is not supported in SQL Server 2008 R2 or earlier.

Demonstration: Filtering Data with TOP and OFFSET-FETCH

In this demonstration, you will see how to filter data using TOP and OFFSET-FETCH.

Demonstration Steps

Filter Data Using TOP and OFFSET-FETCH

1. In Solution Explorer, open the **31 - Demonstration C.sql** script file.
2. In the **Available Databases** list, ensure **ADVENTUREWORKSLT** is selected.
3. Select the code under the comment **Step 1**, and then click **Execute**.
4. Select the code under the comment **Step 2**, and then click **Execute**.
5. Select the code under the comment **Step 3**, and then click **Execute**.
6. Select the code under the comment **Step 4**, and then click **Execute**.
7. Select the code under the comment **Step 5**, and then click **Execute**.
8. Select the code under the comment **Step 6**, and then click **Execute**.
9. Select the code under the comment **Step 7**, and then click **Execute**.
10. Select the code under the comment **Step 8**, and then click **Execute**.
11. Keep SQL Server Management Studio open for the next demonstration.

Check Your Knowledge

Select the best answer

You have a table named Products in your Sales database. You are creating a paged display of products in an application that shows 20 products on each page, ordered by name. Which of the following queries would return the third page of products?

```
SELECT ProductID, ProductName, ProductNumber
FROM Sales.Products
ORDER BY ProductName ASC
OFFSET 60 ROWS FETCH NEXT 20 ROWS ONLY
```

```
SELECT ProductID, ProductName, ProductNumber
FROM Sales.Products
ORDER BY ProductName ASC
OFFSET 40 ROWS FETCH NEXT 20 ROWS ONLY;
```

```
SELECT TOP (20) ProductID, ProductName, ProductNumber
FROM Sales.Products
ORDER BY ProductName ASC
```

```
SELECT TOP (20) WITH TIES ProductID, ProductName, ProductNumber
FROM Sales.Products
ORDER BY ProductName ASC
```

Check answer Show solution Reset

Lesson 4: Working with Unknown Values

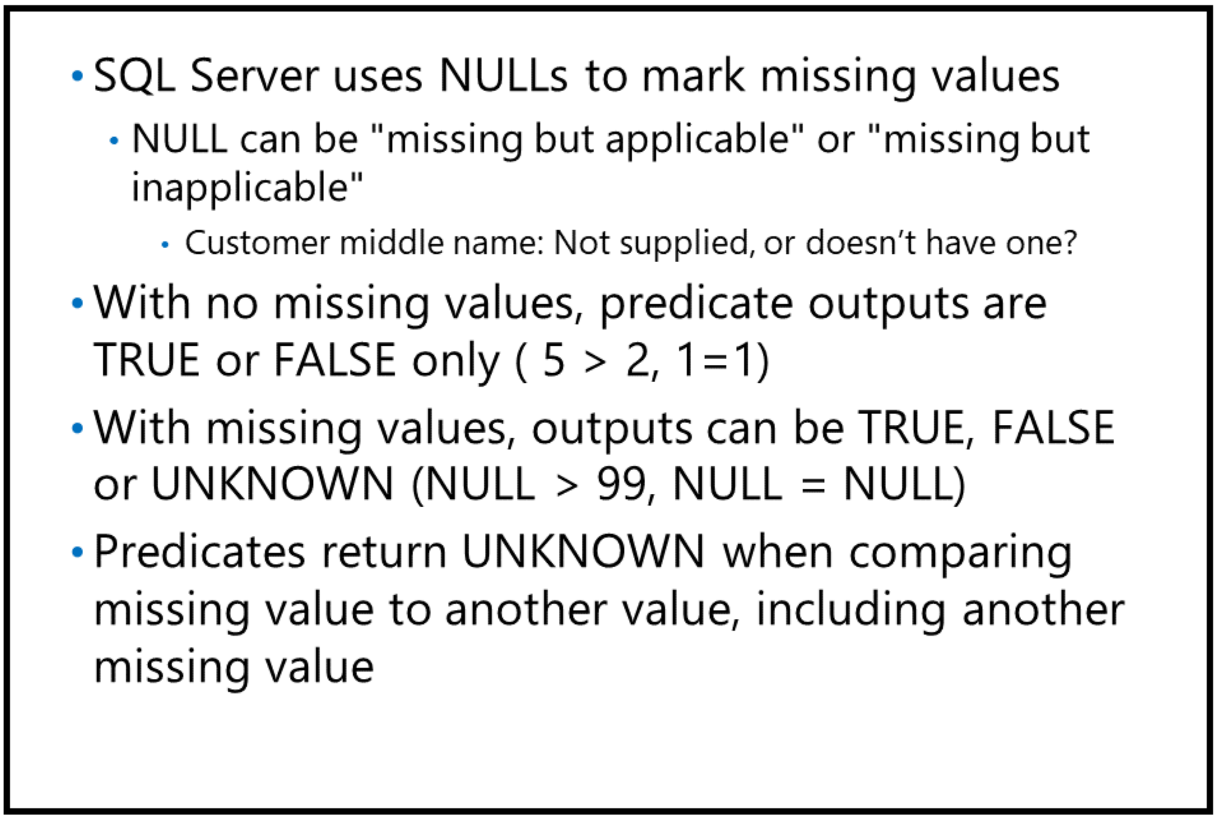
Unlike traditional Boolean logic, predicate logic in SQL Server needs to account for missing values and deal with cases where the result of a predicate is unknown. In this lesson, you will learn how three-valued logic accounts for unknown and missing values; how SQL Server uses NULL to mark missing values; and how to test for NULL in your queries.

Lesson Objectives

After completing this lesson, you will be able to:

- Describe three-valued logic.
- Describe the use of NULL in queries.

Three-Valued Logic



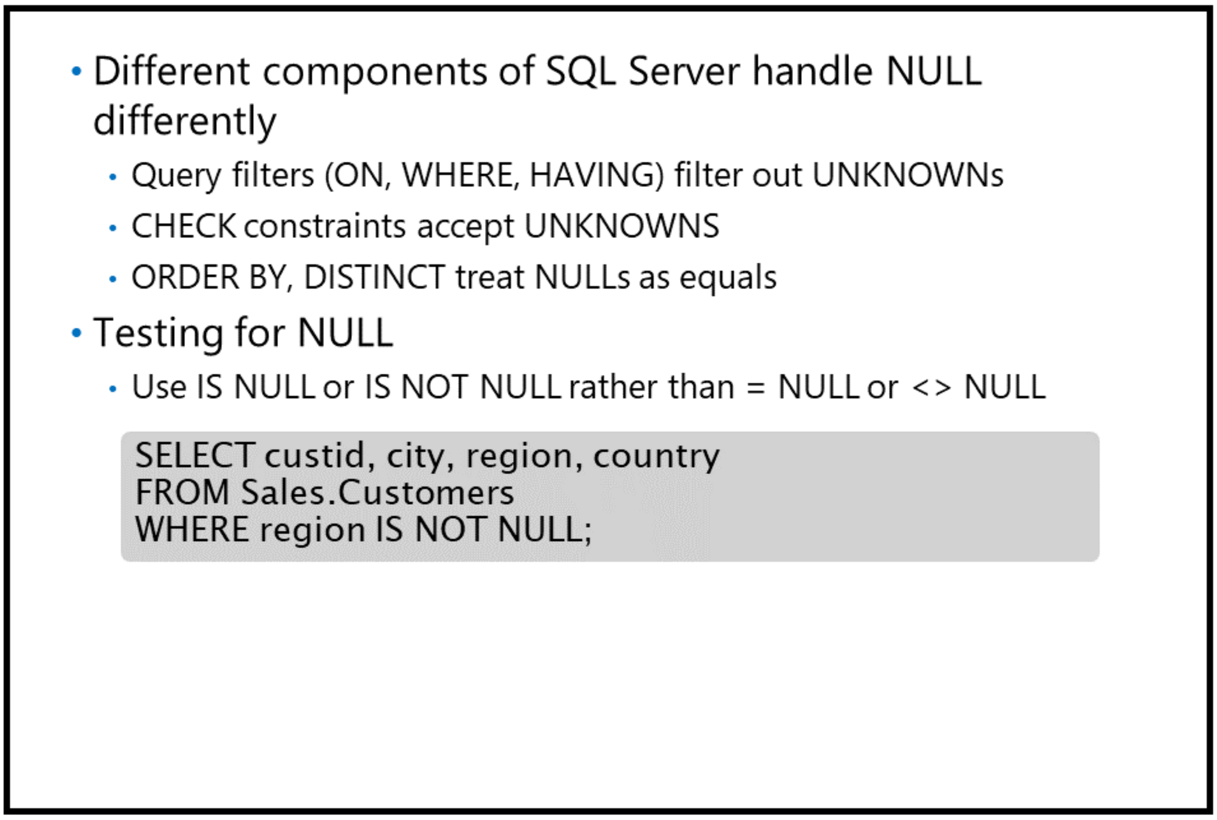
- SQL Server uses NULLs to mark missing values
  - NULL can be "missing but applicable" or "missing but inapplicable"
    - Customer middle name: Not supplied, or doesn't have one?
- With no missing values, predicate outputs are TRUE or FALSE only (5 > 2, 1=1)
- With missing values, outputs can be TRUE, FALSE or UNKNOWN (NULL > 99, NULL = NULL)
- Predicates return UNKNOWN when comparing missing value to another value, including another missing value

Earlier in this course, you learned that SQL Server uses predicate logic as a framework for logical tests that return TRUE or FALSE. This is true for logical expressions where all values being tested are present. If you know the values of both X and Y, you can safely determine whether X>Y is TRUE or FALSE.

However, in SQL Server, not all data being compared may be present. You need to plan for and act on the possibility that some data is missing or unknown. Values in SQL Server may be missing but applicable, such as the value of a middle initial that has not been supplied for an employee. It may also be missing but inapplicable, such as the value of a middle initial for an employee who has no middle name. In both cases, SQL Server will mark the missing value as NULL. A NULL is neither TRUE nor FALSE but is a mark for UNKNOWN, which represents the third value in three-valued logic.

As discussed above, you can determine whether X>Y is TRUE or FALSE when you know the values of both X and Y. But what does SQL Server return for the expression X>Y when Y is missing? SQL Server will return an UNKNOWN, marked as NULL. You will need to account for the possible presence of NULL in your predicate logic, and in the values stored in columns marked with NULL. You will need to write queries that use three-valued logic to account for three possible outcomes—TRUE, FALSE, and UNKNOWN.

Handling NULL in Queries



- Different components of SQL Server handle NULL differently
  - Query filters (ON, WHERE, HAVING) filter out UNKNOWNs
  - CHECK constraints accept UNKNOWNS
  - ORDER BY, DISTINCT treat NULLs as equals
- Testing for NULL
  - Use IS NULL or IS NOT NULL rather than = NULL or <> NULL

```
SELECT custid, city, region, country
FROM Sales.Customers
WHERE region IS NOT NULL;
```

Once you have acquired a conceptual understanding of three-valued logic and NULL, you need to understand the different mechanisms SQL Server uses for handling NULLs. Keep in mind the following guidelines:

- Query filters, such as ON, WHERE, and the HAVING clause, treat NULL like a FALSE result. A WHERE clause that tests for a <column_value> = N will not return rows when the comparison is FALSE. Nor will it return rows when either the column value or the value of N is NULL.

  ORDER BY Query That Includes NULL in Results

```
SELECT empid, lastname, region
FROM HR.Employees
ORDER BY region ASC; --Ascending sort order explicitly included
for clarity.
```

This returns the following, with all employees whose region is missing (marked as NULL) sorted first:

```
empid       lastname             region
----------- -------------------- ---------------
5           Buck                 NULL
6           Suurs                NULL
7           King                 NULL
9           Dolgopyatova         NULL
8           Cameron              WA
1           Davis                WA
2           Funk                 WA
3           Lew                  WA
4           Peled                WA
```

Note: A common question about controlling the display of NULL in queries is whether NULLs can be forced to the end of a result set. As you can see, the ORDER BY clause sorts the NULLs together and first—a behavior you cannot override.

- ORDER BY treats NULLs as if they were the same value and always sorts NULLs together, putting them first in a column. Make sure you test the results of any queries in which the column being used for sort order contains NULLs, and understand the impact of ascending and descending sorts on NULLs.
- In ANSI-compliant queries, a NULL is never equivalent to another value, even another NULL. Queries written to test NULL with an equality will fail to return correct results.

Incorrectly Testing for NULL

```
SELECT empid, lastname, region
FROM HR.Employees
WHERE region = NULL;
```

This returns unexpected results:

```
empid       lastname             region
----------- -------------------- ---------------

(0 row(s) affected)
```

- Use the IS NULL (or IS NOT NULL) operator rather than equal (or not equal).

Correctly Testing for NULL

```
SELECT empid, lastname, region
FROM HR.Employees
WHERE region IS NULL;
```

This returns correct results:

```
empid       lastname             region
----------- -------------------- ---------------
5           Buck                 NULL
6           Suurs                NULL
7           King                 NULL
9           Dolgopyatova         NULL

(4 row(s) affected)
```

Demonstration: Working with NULL

In this demonstration, you will see how to test for NULL.

Demonstration Steps

Test for Null

1. In Solution Explorer, open the **41 - Demonstration D.sql** script file.
2. In the **Available Databases** list, ensure **ADVENTUREWORKSLT** is selected.
3. Select the code under the comment **Step 2**, and then click **Execute**.
4. Select the code under the comment **Step 3**, and then click **Execute**.
5. Select the code under the comment **Step 4**, and then click **Execute**.
6. Select the code under the comment **Step 5**, and then click **Execute**.
7. Select the code under the comment **Step 6**, and then click **Execute**.
8. Select the code under the comment **Step 7**, and then click **Execute**.
9. Close SQL Server Management Studio.

Check Your Knowledge

Discovery

You have the following query:
SELECT e.Name, e.Age
FROM HumanResources.Employees AS e
WHERE YEAR(e.Age) < 1990;
Several employees have asked for their age to be removed from the Human Resources database, and this requested action has been applied to the database. Will the above query return these employees?

Show solution Reset

Lab: Sorting and Filtering Data

Scenario

You are an Adventure Works business analyst who will be writing reports using corporate databases stored in SQL Server. You have been provided with a set of data business requirements and will write T-SQL queries to retrieve the specified data from the databases. You will need to retrieve only some of the available data, and return it to your reports in a specified order.

Objectives

After completing this lab, you will be able to:

- Write queries that filter data using a WHERE clause.
- Write queries that sort data using an ORDER BY clause.
- Write queries that filter data using the TOP option.
- Write queries that filter data using an OFFSET-FETCH clause.

Lab Setup

Estimated Time: 60 minutes

Virtual machine: **20761C-MIA-SQL**

User name: **ADVENTUREWORKS\Student**

Password: **Pa55w.rd**

Exercise 1: Write Queries that Filter Data Using a WHERE Clause

Scenario

The marketing department is working on several campaigns for existing customers and staff need to obtain different lists of customers, depending on several business rules. Based on these rules, you will write the SELECT statements to retrieve the needed rows from the Sales.Customers table.

The main tasks for this exercise are as follows:

1. Prepare the Lab Environment
2. Write a SELECT Statement Using a WHERE Clause
3. Write a SELECT Statement Using an IN Predicate in the WHERE Clause
4. Write a SELECT Statement Using a LIKE Predicate in the WHERE Clause
5. Observe the T-SQL Statement Provided by the IT Department
6. Write a SELECT Statement to Retrieve Customers Without Orders

Detailed Steps ▼

Detailed Steps ▼

Detailed Steps ▼

Detailed Steps ▼

Detailed Steps ▼

Detailed Steps ▼

Result: After this exercise, you should be able to filter rows of data from one or more tables by using WHERE predicates with logical operators.

Exercise 2: Write Queries that Sort Data Using an ORDER BY Clause

Scenario

The sales department would like a report showing all the orders with some customer information. An additional request is that the result be sorted by the order dates and the customer IDs. From previous modules, remember that the order of the rows in the output of a query without an ORDER BY clause is not guaranteed. Because of this, you will have to write a SELECT statement that uses an ORDER BY clause.

The main tasks for this exercise are as follows:

1. Write a SELECT Statement Using an ORDER BY Clause
2. Apply the Needed Changes and Execute the T-SQL Statement
3. Order the Result by the firstname Column

Detailed Steps ▼

Detailed Steps ▼

Detailed Steps ▼

Result: After this exercise, you should know how to use an ORDER BY clause.

Exercise 3: Write Queries that Filter Data Using the TOP Option

Scenario

The sales department wants to have some additional reports that show the last invoiced orders and the top 10 percent of the most expensive products being sold.

The main tasks for this exercise are as follows:

1. Writing Queries That Filter Data Using the TOP Clause
2. Use the OFFSET-FETCH Clause to Implement the Same Task
3. Write a SELECT Statement to Retrieve the Most Expensive Products

Detailed Steps ▼

Detailed Steps ▼

Detailed Steps ▼

Result: After this exercise, you should have an understanding of how to apply the TOP option in the SELECT clause of a T-SQL statement.

Exercise 4: Write Queries that Filter Data Using the OFFSET-FETCH Clause

Scenario

In this exercise, you will implement a paging solution for displaying rows from the **Sales.Orders** table because the total number of rows is high. In each page of a report, the user should only see 20 rows.

The main tasks for this exercise are as follows:

1. OFFSET-FETCH Clause to Fetch the First 20 Rows
2. Use the OFFSET-FETCH Clause to Skip the First 20 Rows
3. Write a Generic Form of the OFFSET-FETCH Clause for Paging

Detailed Steps ▼

Detailed Steps ▼

Detailed Steps ▼

Result: After this exercise, you will be able to use OFFSET-FETCH to work page-by-page through a result set returned by a SELECT statement.

Review Question(s)

Check Your Knowledge

Discovery

What is the difference between filtering using the TOP option, and filtering using the WHERE clause?

Show solution Reset

Module Review and Takeaways

In this module, you have learned how to enhance a query to limit the number of rows that the query returns, and control the order in which the rows are displayed.

Review Question(s)

Check Your Knowledge

Discovery

Does the physical order of rows in a SQL Server table guarantee any sort order in queries using the table?

Show solution Reset

Check Your Knowledge

Discovery

You have the following query:

SELECT p.PartNumber, p.ProductName, o.Quantity

FROM Sales.Products AS p

LEFT OUTER JOIN Sales.OrderItems AS o

ON p.ID = o.ProductID

ORDER BY o.Quantity ASC

You have one new product that has yet to receive any orders. Will this product appear at the top or the bottom of the results?

Show solution Reset