MICROSOFT
TRAINING
AND CERTIFICATION

Microsoft Official Curriculum

Module 7: Creating and Maintaining Indexes

Contents

Microsoft

Project Lead: Rich Rose
Instructional Designers: Rich Rose, Cheryl Hoople, Marilyn McGill
Instructional Software Design Engineers: Karl Dehmer, Carl Raebler, Rick Byham
Technical Lead: Karl Dehmer
Subject Matter Experts: Karl Dehmer, Carl Raebler, Rick Byham
Graphic Artist: Kirsten Larson (Independent Contractor)
Editing Manager: Lynette Skinner
Editor: Wendy Cleary
Copy Editor: Edward McKillop (S&T Consulting)
Production Manager: Miracle Davis
Production Coordinator: Jenny Boe
Production Support: Lori Walker (S&T Consulting)
Test Manager: Sid Benavente
Courseware Testing: TestingTesting123
Classroom Automation: Lorrin Smith-Bates
Creative Director, Media/Sim Services: David Mahlmann
Web Development Lead: Lisa Pease
CD Build Specialist: Julie Challenger
Online Support: David Myka (S&T Consulting)
Localization Manager: Rick Terek
Operations Coordinator: John Williams
Manufacturing Support: Laura King; Kathy Hershey
Lead Product Manager, Release Management: Bo Galford
Lead Product Manager, Data Base: Margo Crandall
Group Manager, Courseware Infrastructure: David Bramble
Group Product Manager, Content Development: Dean Murray
General Manager: Robert Stewart

Instructor Notes

Presentation:
60 Minutes

Lab:
60 Minutes

This module provides students with an overview of creating and maintaining indexes with the CREATE INDEX options. It describes how maintenance procedures physically change indexes. The module discusses maintenance tools and describes the use of statistics in Microsoft® SQL Server™ 2000. It also describes ways to verify that indexes are used, and explains how to tell whether they are performing optimally. The module concludes with a discussion of when to use the Index Tuning Wizard.

After completing this module, students will be able to:

- Create indexes and indexed views with unique or composite characteristics.
- Use the CREATE INDEX options.
- Describe how to maintain indexes over time.
- Describe how the query optimizer creates, stores, maintains, and uses statistics to optimize queries.
- Query the **sysindexes** table.
- Describe how the Index Tuning Wizard works and when to use it.
- Describe performance considerations that affect creating and maintaining indexes.

Materials and Preparation

This section provides the materials and preparation tasks that you need to teach this module.

Required Materials

To teach this module, you need the following materials:

- Microsoft PowerPoint® file 2073A_07.ppt
- The C:\Moc\2073A\Demo\D07_Ex.sql example file, which contains all of the example scripts from the module, unless otherwise noted in the module.

Preparation Tasks

To prepare for this module, you should:

- Read all of the materials for this module.
- Complete the labs.

Module Strategy

Use the following strategy to present this module:

- Creating Indexes

  Discuss the CREATE INDEX and DROP INDEX statements and explain the benefits of defining indexes with unique or composite characteristics. Explain that obtaining information on existing indexes is recommended before students create, modify, or delete them.

- Creating Index Options

  Explain the use of the FILLFACTOR option. Emphasize the importance of using it with the PAD_INDEX option to optimize insert and update performance over time on tables that contain indexes.

- Maintaining Indexes

  Explain that indexes must be maintained over time because data becomes fragmented as tables grow. Discuss how the DBCC SHOWCONTIG statement helps to maintain index performance. Then, point out that the DROP_EXISTING option is used to maintain existing indexes. This option changes the index definition and accelerates the rebuilding of indexes.

- Introduction to Statistics

  Discuss how the query optimizer uses statistics to determine whether an index is useful. Explain how statistics are gathered, stored, created, updated, and viewed. Emphasize the importance of having current statistics.

- Querying the **sysindexes** Table

  Discuss how to query the **sysindexes** table to get table and index information, in addition to statistics for each index. Review the partial list of the information that a query can return.

- Setting Up Indexes Using the Index Tuning Wizard

  Present the Index Tuning Wizard and explain when to use the wizard. Review the facts and guidelines to consider when using it.

- Performance Considerations

  Discuss the performance considerations that affect creating and maintaining indexes.

Customization Information

This section identifies the lab setup requirements for a module and the configuration changes that occur on student computers during the labs. This information is provided to assist you in replicating or customizing Microsoft Official Curriculum (MOC) courseware.

Important The labs in this module are dependent on the classroom configuration that is specified in the Customization Information section at the end of the *Classroom Setup Guide* for course 2073A, *Programming a Microsoft SQL Server 2000 Database*.

Lab Setup

The following section describes the setup requirement for the labs in this module.

Setup Requirement 1

The lab in this module requires the **ClassNorthwind** database to be in a state required for this lab. To prepare student computers to meet this requirement, perform one of the following actions:

- Complete the prior lab
- Execute the C:\Moc\2073A\Batches\Restore07A.cmd batch file.

Setup Requirement 2

The lab in this module requires the **ClassNorthwind** database to be in a state required for this lab. To prepare student computers to meet this requirement, perform one of the following actions:

- Complete the prior lab
- Execute the C:\Moc\2073A\Batches\Restore07B.cmd batch file.

Warning If this course has been customized, students must execute the C:\Moc\2073A\Batches\Restore07A.cmd batch file to ensure that the first lab will function properly.

If this course has been customized, students must execute the C:\Moc\2073A\Batches\Restore07B.cmd batch file to ensure that the second lab will function properly.

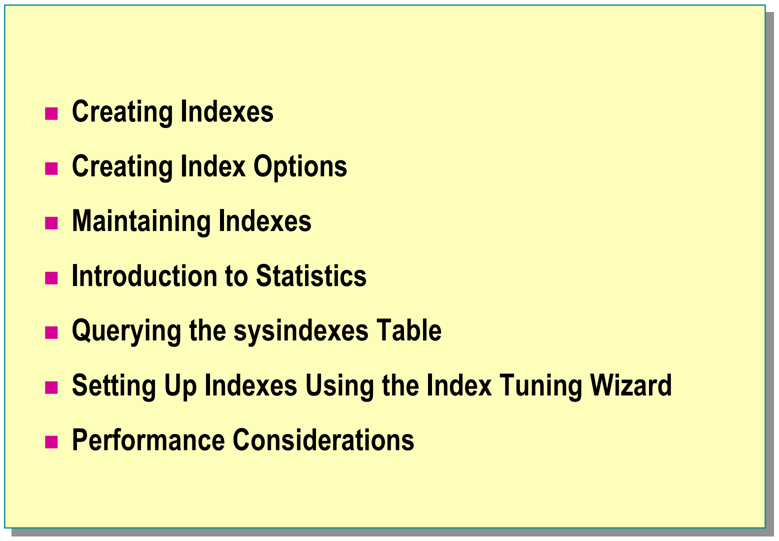
Lab Results

There are no configuration changes on student computers that affect replication or customization.

Overview

Topic Objective
To provide an overview of the module topics and objectives.

Lead-in
In this module, you will learn about creating and maintaining indexes.



- **Creating Indexes**
- **Creating Index Options**
- **Maintaining Indexes**
- **Introduction to Statistics**
- **Querying the sysindexes Table**
- **Setting Up Indexes Using the Index Tuning Wizard**
- **Performance Considerations**

When you program a database, you want to create useful indexes that enable you to quickly gain access to data. By using Microsoft® Windows® 2000, you can create and maintain indexes and statistics. When you use the Index Tuning Wizard, Microsoft SQL Server™ 2000 creates indexes, analyzes your queries, and determines the indexes that you should create.

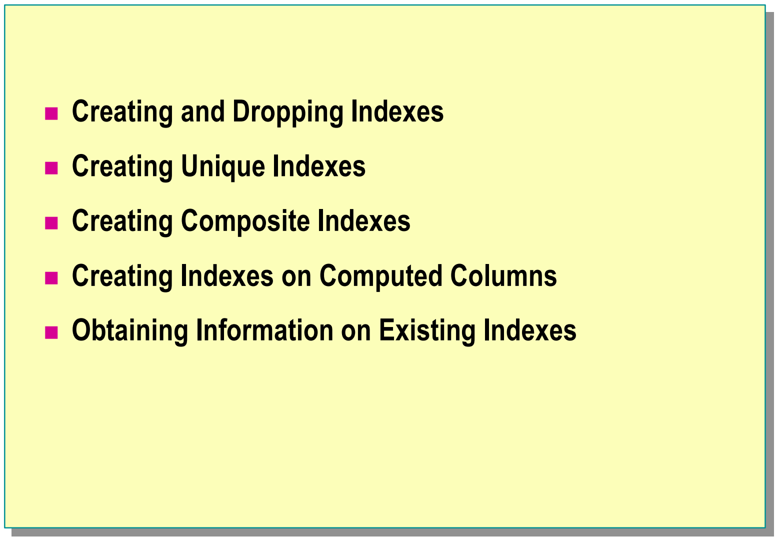
After completing this module, you will be able to:

- Create indexes and indexed views with unique or composite characteristics.
- Use the CREATE INDEX options.
- Describe how to maintain indexes over time.
- Describe how the query optimizer creates, stores, maintains, and uses statistics to optimize queries.
- Query the **sysindexes** table.
- Describe how the Index Tuning Wizard works and when to use it.
- Describe performance considerations that affect creating and maintaining indexes.

◆ Creating Indexes

Topic Objective
To introduce the topics on creating indexes.

Lead-in
Now that you're familiar with the different index architectures, let's discuss creating indexes and obtaining information on existing indexes.



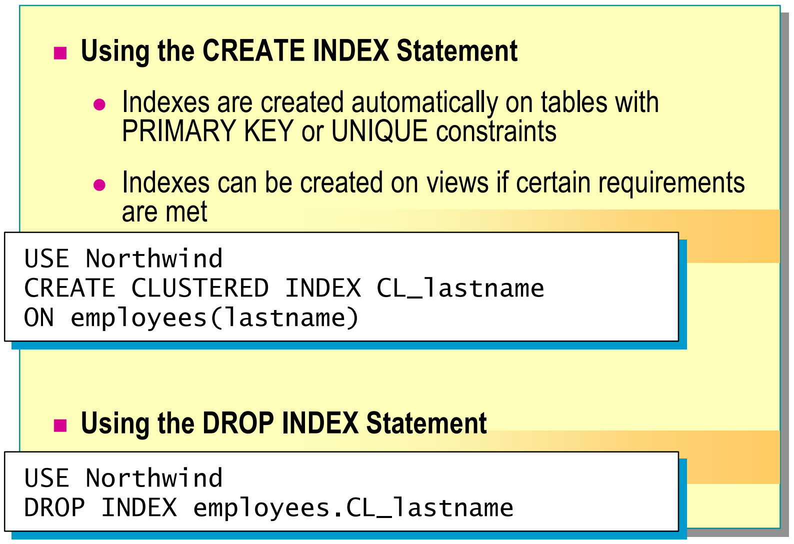
- Creating and Dropping Indexes
- Creating Unique Indexes
- Creating Composite Indexes
- Creating Indexes on Computed Columns
- Obtaining Information on Existing Indexes

Now that you are familiar with the different index architectures, we will discuss creating and dropping indexes and obtaining information on existing indexes.

Creating and Dropping Indexes

Topic Objective
To introduce the CREATE INDEX and DROP INDEX statements.

Lead-in
You create indexes by using the CREATE INDEX statement and can remove them by using the DROP INDEX statement.



- **Using the CREATE INDEX Statement**
  - Indexes are created automatically on tables with PRIMARY KEY or UNIQUE constraints
  - Indexes can be created on views if certain requirements are met

```
USE Northwind
CREATE CLUSTERED INDEX CL_lastname
ON employees(lastname)
```

- **Using the DROP INDEX Statement**

```
USE Northwind
DROP INDEX employees.CL_lastname
```

You create indexes by using the CREATE INDEX statement and can remove them by using the DROP INDEX statement.

Note You must be the table owner to execute either statement in a database.

Using the CREATE INDEX Statement

Use the CREATE INDEX statement to create indexes. You also can use the Create Index Wizard in SQL Server Enterprise Manager. When you create an index on one or more columns in a table, consider the following facts and guidelines:

- SQL Server automatically creates indexes when a PRIMARY KEY or UNIQUE constraint is created on a table. Defining a PRIMARY KEY or UNIQUE constraint is preferred over creating standard indexes.
- You must be the table owner to execute the CREATE INDEX statement.
- Indexes can be created on views.
- SQL Server stores index information in the **sysindexes** system table.
- Before you create an index on a column, determine whether indexes already exist on that column.
- Keep your indexes small by defining them on columns that are small in size. Typically, smaller indexes are more efficient than indexes with larger key values.
- Select columns on the basis of uniqueness so that each key value identifies a small number of rows.
- When you create a clustered index, all existing nonclustered indexes are rebuilt.

Syntax

CREATE [UNIQUE] [CLUSTERED | NONCLUSTERED]
  INDEX *index_name* ON { *table* | *view* } (*column* [ASC | DESC] [,...*n*])
  [WITH
  [PAD_INDEX]
  [[,] FILLFACTOR = *fillfactor*]
  [[,] IGNORE_DUP_KEY]
  [[,] DROP_EXISTING]
  [[,] STATISTICS_NORECOMPUTE]
  [[,] SORT_IN_TEMPDB]
  ]
  [ON *filegroup*]

Example 1

This example creates a clustered index on the **LastName** column in the **Employees** table.

```
CREATE CLUSTERED INDEX CL_lastname
  ON employees(lastname)
```

Using the DROP INDEX Statement

Use the DROP INDEX statement to remove an index on a table. When you drop an index, consider the following facts:

- SQL Server reclaims disk space that is occupied by the index when you execute the DROP INDEX statement.
- You cannot use the DROP INDEX statement on indexes that are created by PRIMARY KEY or UNIQUE constraints. You must drop the constraint in order to drop these indexes.
- When you drop a table, all indexes for that table are also dropped.
- When you drop a clustered index, all nonclustered indexes on the table are rebuilt automatically.
- You must be in the database in which an index resides in order to drop that index.
- The DROP INDEX statement cannot be used on system tables.

Syntax

DROP INDEX '*table.index* | *view.index*' [, ...*n*]

Example 2

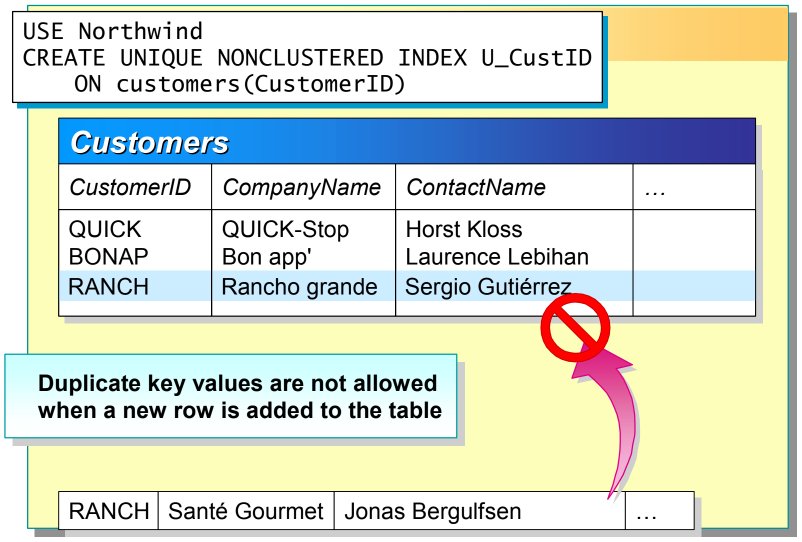
This example drops the **cl_lastname** index from the **Member** table.

```
USE Northwind
DROP INDEX employees.CL_lastname
```

Creating Unique Indexes

Topic Objective
To explain how unique indexes ensure that indexed columns are unique.

Lead-in
In a unique index, no two rows are permitted to have the same key value.



```
USE Northwind
CREATE UNIQUE NONCLUSTERED INDEX U_CustID
    ON customers(CustomerID)
```

Customers

CustomerID	*CompanyName*	*ContactName*	…
QUICK	QUICK-Stop	Horst Kloss	
BONAP	Bon app'	Laurence Lebihan	
RANCH	Rancho grande	Sergio Gutiérrez	

Duplicate key values are not allowed when a new row is added to the table

RANCH	Santé Gourmet	Jonas Bergulfsen	…

A *unique* index ensures that all data in an indexed column is unique and does not contain duplicate values.

Unique indexes ensure that data in indexed columns is unique. If the table has a PRIMARY KEY or UNIQUE constraint, SQL Server automatically creates a unique index when you execute the CREATE TABLE or ALTER TABLE statement.

Ensuring That Data in Indexed Columns Is Unique

Create a unique index for clustered or nonclustered indexes when the data itself is inherently unique.

However, if uniqueness must be enforced, create PRIMARY KEY or UNIQUE constraints on the column rather than creating a unique index. When you create a unique index, consider the following facts and guidelines:

- SQL Server automatically creates unique indexes on columns in a table defined with PRIMARY KEY or UNIQUE constraints.
- If a table contains data, SQL Server checks for duplicate values when you create the index.
- SQL Server checks for duplicate values each time that you use the INSERT or UPDATE statement. If duplicate key values exist, SQL Server cancels your statement and returns an error message with the first duplicate.
- Ensure that each row has a unique value—no two rows can have the same identification number if a unique index is created on that column. This regulation ensures that each entity is identified uniquely.
- Create unique indexes only on columns in which entity integrity can be enforced. For example, you would not create a unique index on the **LastName** column of the **Employees** table because some employees may have the same last names.

Example 1

This example creates a unique, nonclustered index named **U_CustID** on the **Customers** table. The index is built on the **CustomerID** column. The value in the **CustomerID** column must be a unique value for each row of the table.

```
USE Northwind
CREATE UNIQUE NONCLUSTERED INDEX U_CustID
  ON customers(CustomerID)
```

Finding All Duplicate Values in a Column

If duplicate key values exist when you create a unique index, the CREATE INDEX statement fails. SQL Server returns an error message with the first duplicate, but other duplicate values may exist, as well. Use the following sample script on any table to find all duplicate values in a column. Replace the italicized text with information specific to your query.

```
SELECT index_col, COUNT (index_col)
FROM tablename
GROUP BY index_col
HAVING COUNT(index_col)>1 ORDER BY index_col
```

Example 2

This example determines whether duplicate customer identification exists in the **CustomerID** column in the **Customers** table. If so, SQL Server returns the customer identification and number of duplicate entries in the result set.

```
SELECT CustomerID, COUNT(CustomerID) AS '# of Duplicates'
FROM Northwind.dbo.Customers
GROUP BY CustomerID
HAVING COUNT(CustomerID)>1
ORDER BY CustomerID
```

Result

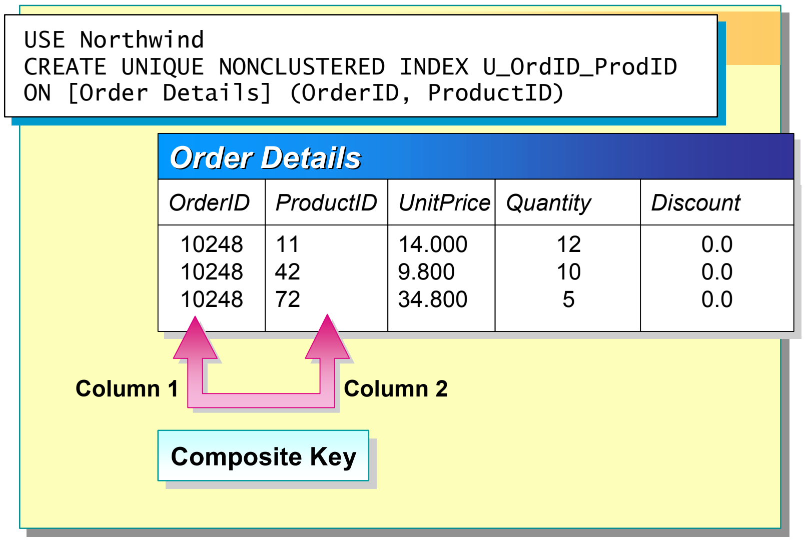
```
CustomerID           # of Duplicates

(0 row(s) affected)
```

Creating Composite Indexes

Topic Objective
To explain how composite indexes work.

Lead-in
Composite indexes specify more than one column as the key value.



```
USE Northwind
CREATE UNIQUE NONCLUSTERED INDEX U_OrdID_ProdID
ON [Order Details] (OrderID, ProductID)
```

Order Details

OrderID	*ProductID*	*UnitPrice*	*Quantity*	*Discount*
10248	11	14.000	12	0.0
10248	42	9.800	10	0.0
10248	72	34.800	5	0.0

Delivery Tip
Point out that the **OrderID** column is the first column because it is more selective than the **ProductID** column.

Composite indexes specify more than one column as the key value. You create composite indexes:

- When two or more columns are best searched as a key.
- If queries reference only the columns in the index.

For example, a telephone directory is a good example of where a composite index would be useful. The directory is organized by last names. Within the last names, it is organized by first names, because entries with the same last name often exist.

When you create a composite index, consider the following facts and guidelines:

- You can combine as many as 16 columns into a single composite index. The sum of the lengths of the columns that make up the composite index cannot exceed 900 bytes.
- All columns in a composite index must be from the same table, except when an index is created on a view.
- Define the most unique column first. The first column defined in the CREATE INDEX statement is referred to as the *highest order*.
- The WHERE clause of a query must reference the first column of the composite index for the query optimizer to use the composite index.
- An index on (**column1**, **column2**) is not the same as an index on (**column2**, **column1**)—each has a distinct column order. The column that contains more selective data or that would return the lowest percentage of rows often determines the column order.

- Composite indexes are useful for tables with multiple column keys.
- Use composite indexes to increase query performance and reduce the number of indexes that you create on a table.

Note Multiple indexes on the same columns are typically not useful.

Example

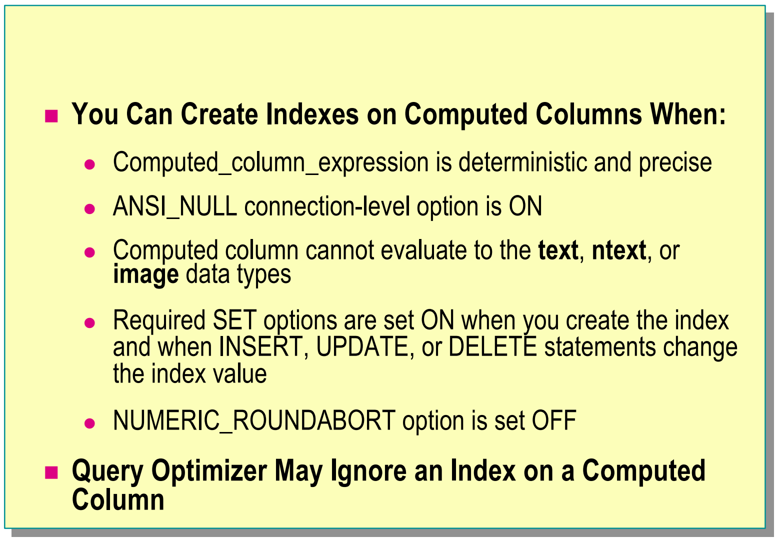
This example creates a nonclustered, composite index on the **Order Details** table. The **OrderID** and the **ProductID** columns are the composite key values. Notice that the **OrderID** column is listed first because it is more selective than the **ProductID** column.

```
USE Northwind
CREATE UNIQUE NONCLUSTERED INDEX U_OrdID_ProdID
ON [Order Details] (OrderID, ProductID)
```

Creating Indexes on Computed Columns

Topic Objective
To explain the requirements for creating an index on a computed column.

Lead-in
Before you can create an index on a computed column, you must meet several requirements.



- **You Can Create Indexes on Computed Columns When:**
  - Computed_column_expression is deterministic and precise
  - ANSI_NULL connection-level option is ON
  - Computed column cannot evaluate to the **text**, **ntext**, or **image** data types
  - Required SET options are set ON when you create the index and when INSERT, UPDATE, or DELETE statements change the index value
  - NUMERIC_ROUNDABORT option is set OFF
- **Query Optimizer May Ignore an Index on a Computed Column**

For Your Information
Point out that SQL Server Books Online contains additional information on deterministic and nondeterministic functions.

For Your Information
Point out that SQL Server Books Online contains additional information on SET options that affect results.

You can create indexes on computed columns when:

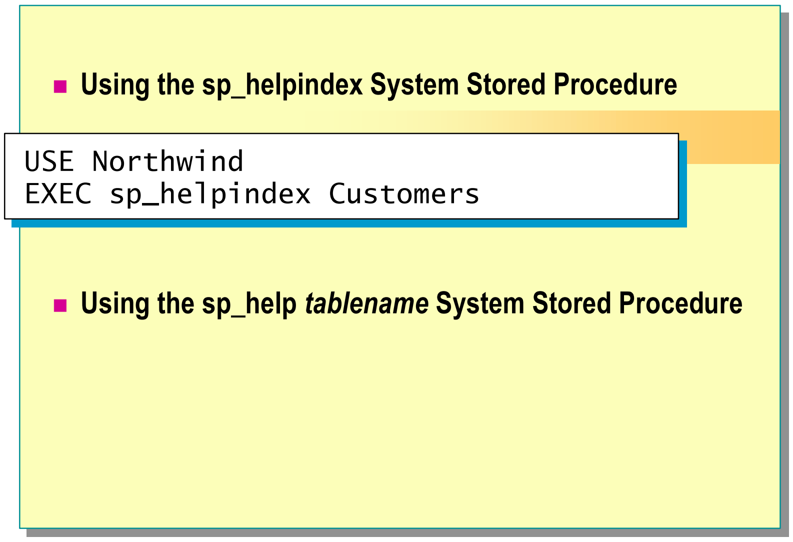
- The computed_column_expression is *deterministic*. Deterministic expressions always return the same result.
- The ANSI_NULL connection-level option is ON when the CREATE TABLE statement is executed. The OBJECTPROPERTY function reports whether the option is on through the **IsAnsiNullsOn** property.
- The computed_column_expression that is defined for the computed column cannot evaluate the **text**, **ntext**, or **image** data types.
- The connection on which the index is created, and all connections attempting INSERT, UPDATE, or DELETE statements that will change values in the index, have six SET options set ON and one option set OFF. These options must be set on:
  - ANSI_NULLS
  - ANSI_PADDING
  - ANSI_WARNINGS
  - ARITHABORT
  - CONCAT_NULL_YIELDS_NULL
  - QUOTED_IDENTIFIER
- In addition to these ON settings, the NUMERIC_ROUNDABORT option must be set OFF.

Note The query optimizer ignores an index on a computed column for any SELECT statement that is executed by a connection that does not have these same option settings.

Obtaining Information on Existing Indexes

Topic Objective
To discuss obtaining information on existing indexes.

Lead-in
You may require information about existing indexes before you create, modify, or remove an index.



- **Using the sp_helpindex System Stored Procedure**

```
USE Northwind
EXEC sp_helpindex Customers
```

- **Using the sp_help *tablename* System Stored Procedure**

You may require information about existing indexes before you create, modify, or remove an index.

Using the sp_helpindex System Stored Procedure

You can use SQL Server Enterprise Manager or execute the **sp_helpindex** system stored procedure to obtain index information, such as index name, type, and options for a specific table.

Example

This example lists the indexes on the **Customers** table.

```
USE Northwind
EXEC sp_helpindex Customers
```

Result

```
index_name        index_description                                index_keys
PK_Customers      clustered, unique, Primary Key located on PRIMARY CustomerID
PostalCode        nonclustered located on PRIMARY                  PostalCode
City              nonclustered located on PRIMARY                  City

(1 row(s) affected)
```

Delivery Tip
Demonstrate the result set of **sp_help** *tablename* by executing the following:

USE Northwind
EXEC sp_help Customers

Point out the additional table information.

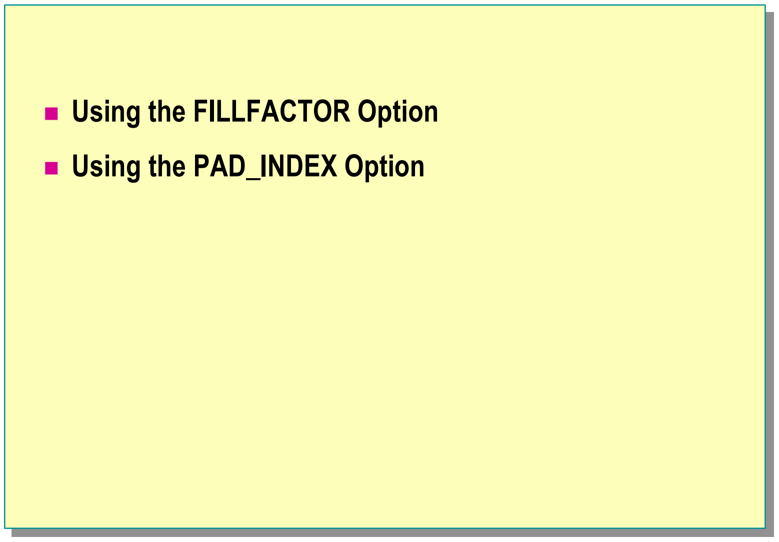
Using the sp_help *tablename* System Stored Procedure

You can also execute the **sp_help** *tablename* system stored procedure to obtain information on indexes, as well as other table information.

◆ Creating Index Options

Topic Objective
To provide a brief overview of some of the creating index options.

Lead-in
SQL Server offers creating index options that can speed up index creation and also enhance index performance over time.



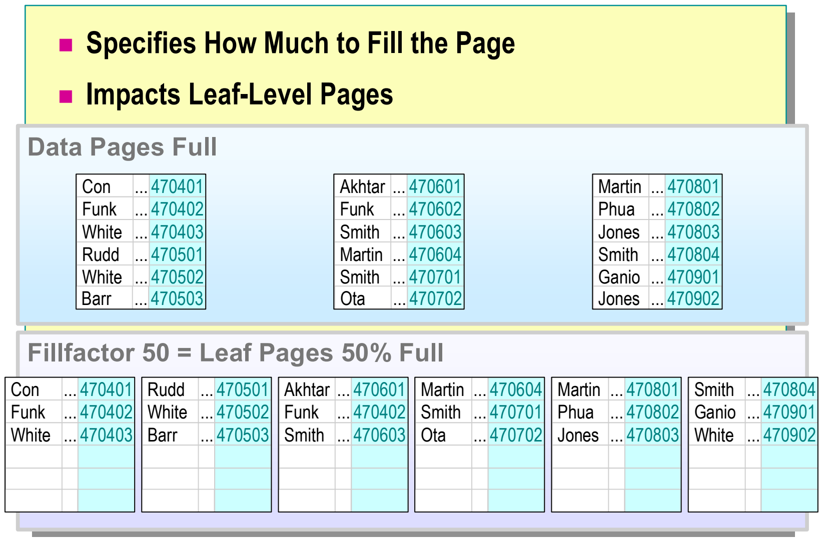
- **Using the FILLFACTOR Option**
- **Using the PAD_INDEX Option**

SQL Server offers creating index options that can speed up index creation and also enhance index performance over time.

Using the FILLFACTOR Option

Topic Objective
To determine when and how to use FILLFACTOR options to improve performance.

Lead-in
You can use the FILLFACTOR option to optimize the performance of INSERT and UPDATE statements on tables that contain clustered or nonclustered indexes.



- **Specifies How Much to Fill the Page**
- **Impacts Leaf-Level Pages**

Data Pages Full

Con ... 470401	Akhtar ... 470601	Martin ... 470801
Funk ... 470402	Funk ... 470602	Phua ... 470802
White ... 470403	Smith ... 470603	Jones ... 470803
Rudd ... 470501	Martin ... 470604	Smith ... 470804
White ... 470502	Smith ... 470701	Ganio ... 470901
Barr ... 470503	Ota ... 470702	Jones ... 470902

Fillfactor 50 = Leaf Pages 50% Full

Con ... 470401	Rudd ... 470501	Akhtar ... 470601	Martin ... 470604	Martin ... 470801	Smith ... 470804
Funk ... 470402	White ... 470502	Funk ... 470402	Smith ... 470701	Phua ... 470802	Ganio ... 470901
White ... 470403	Barr ... 470503	Smith ... 470603	Ota ... 470702	Jones ... 470803	White ... 470902

You can use the FILLFACTOR option to optimize the performance of INSERT and UPDATE statements on tables that contain clustered or nonclustered indexes.

When an index page becomes full, SQL Server must take time to split the page to make room for new rows. Use the FILLFACTOR option to allocate a percentage of free space on the leaf-level index pages to reduce page splitting.

Note The FILLFACTOR option is applied only when the index is created or rebuilt. SQL Server does not dynamically maintain the specified percentage of allocated space on the index pages.

The fillfactor value that you specify on a table depends on how often data is modified (INSERT and UPDATE statements) and your organization's environment. Generally, you should:

- Use a low fillfactor value for online transaction processing (OLTP) environments.
- Use a high fillfactor value for SQL Server Analysis Services environments.

The following table shows the FILLFACTOR option settings and the typical environments in which these fillfactor values are used.

FILLFACTOR percentage	Leaf-level pages	Non-leaf-level pages	Activity on key values	Typical business environment
0 (default)	Fill completely	Leave room for one index entry	None to light modification	Analysis Services
1–99	Fill to specified percentage	Leave room for one index entry	Moderate to heavy modification	Mixed or OLTP
100	Fill completely	Leave room for one index entry	None to light modification	Analysis Services

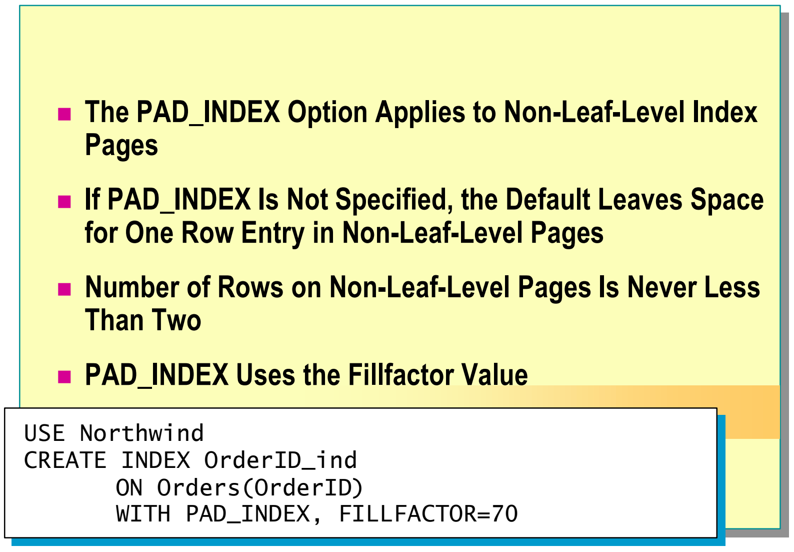
When you use the FILLFACTOR option, consider the following facts and guidelines:

- Fillfactor values range from 1 to 100 percent.
- The default fillfactor value is 0. This value fills the leaf-level index pages to 100 percent and leaves room for the maximum size of one index entry in the non-leaf-level index pages. You cannot explicitly specify fillfactor = 0.
- You can change the default fillfactor value at the server level by using the **sp_configure** system stored procedure.
- The **sysindexes** system table stores the fillfactor value that was last applied, along with other index information.
- The fillfactor value is specified in percentages. The percentage determines how much the leaf-level pages should be filled. For example, a fillfactor of 65 percent fills the leaf-level pages 65 percent, leaving 35 percent of the page space free for new rows. The size of the row
  has an impact on how many rows can fit into or fill the page for the specified fillfactor percentage.
- Use the FILLFACTOR option on tables into which many rows are inserted, or when clustered index key values are frequently modified.

Using the PAD_INDEX Option

Topic Objective
To discuss using the PAD_INDEX option.

Lead-in
You can use the PAD_INDEX option with the FILLFACTOR option to optimize performance when you create or rebuild an index.



- **The PAD_INDEX Option Applies to Non-Leaf-Level Index Pages**
- **If PAD_INDEX Is Not Specified, the Default Leaves Space for One Row Entry in Non-Leaf-Level Pages**
- **Number of Rows on Non-Leaf-Level Pages Is Never Less Than Two**
- **PAD_INDEX Uses the Fillfactor Value**

```
USE Northwind
CREATE INDEX OrderID_ind
       ON Orders(OrderID)
       WITH PAD_INDEX, FILLFACTOR=70
```

Delivery Tip
Point out that students will rarely use the PAD_INDEX option. Most performance gains come from using FILLFACTOR appropriately.

The PAD_INDEX option specifies the percentage to which to fill the non-leaf-level index pages. You can use the PAD_INDEX option only when FILLFACTOR is specified, because the PAD_INDEX percentage value is determined by the percentage value specified for FILLFACTOR.

The following table shows the impact of FILLFACTOR option settings when you use the PAD_INDEX option, and the typical environment in which PAD_INDEX values are used.

FILLFACTOR percentage	Leaf-level pages	Non-leaf-level pages	Activity on key values	Typical business environment
1–99	Fill to specified percentage	Fill to specified percentage	Moderate to heavy modification	OLTP

When you use the PAD_INDEX option, consider the following facts:

- SQL Server applies the percentage that the FILLFACTOR option specifies to the leaf-level and non-leaf-level pages.
- By default, SQL Server always leaves enough room to accommodate at least one row of the maximum index size for each non-leaf-level page, regardless of how high the fillfactor value is.
- The number of items on the non-leaf-level index page is never fewer than two, regardless of how low the fillfactor value is.
- PAD_INDEX uses the fillfactor value.

Example

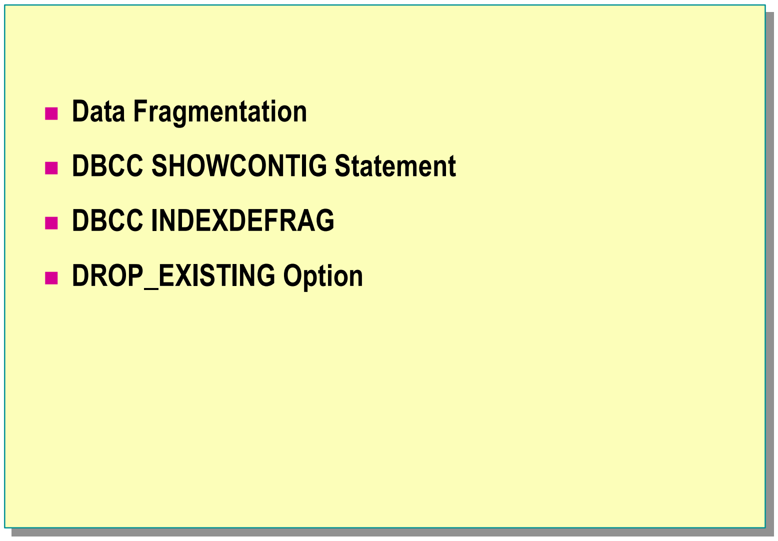
This example creates the **OrderID_ind** index on the **OrdersID** column in the **Orders** table. By specifying the PAD_INDEX option with the FILLFACTOR option, SQL Server creates leaf-level and non-leaf-level pages that are 70 percent full. However, if you do not use the PAD_INDEX option, the leaf-level pages are 70 percent full, and the non-leaf-level pages are almost completely filled.

```
USE Northwind
CREATE INDEX OrderID_ind
  ON Orders(OrderID)
  WITH PAD_INDEX, FILLFACTOR=70
```

◆ Maintaining Indexes

Topic Objective
To provide a brief overview of several maintenance tools.

Lead-in
You must maintain indexes after you create them to ensure optimal performance.



- **Data Fragmentation**
- **DBCC SHOWCONTIG Statement**
- **DBCC INDEXDEFRAG**
- **DROP_EXISTING Option**

You must maintain indexes after you create them to ensure optimal performance. Over time, data becomes fragmented. You manage data fragmentation according to your organization's environment.

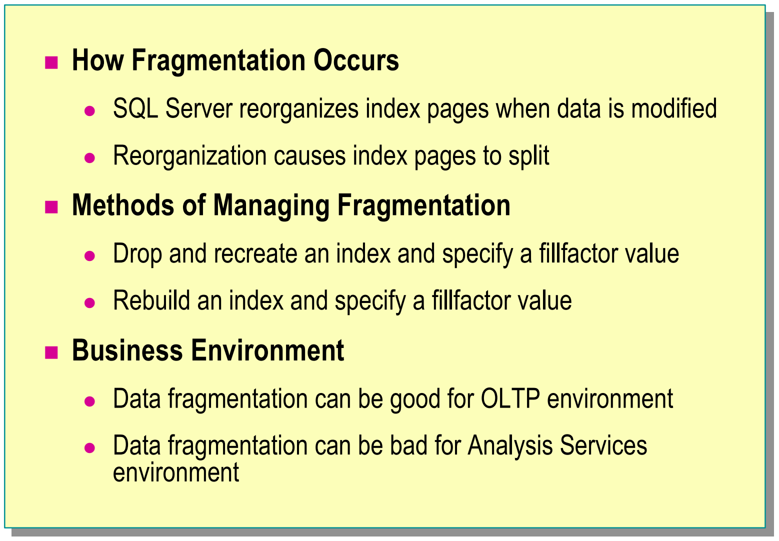
SQL Server provides an Index Tuning Wizard that tracks the usage of your indexes automatically and assists with maintaining and creating indexes that perform optimally.

You also can use various options and tools to help you rebuild indexes and verify index optimization.

Data Fragmentation

Topic Objective
To discuss how data fragmentation occurs and ways to manage it.

Lead-in
Over time, fragmentation occurs as tables are modified. You apply different methods to manage the fragmentation based on your business environment.



- **How Fragmentation Occurs**
  - SQL Server reorganizes index pages when data is modified
  - Reorganization causes index pages to split
- **Methods of Managing Fragmentation**
  - Drop and recreate an index and specify a fillfactor value
  - Rebuild an index and specify a fillfactor value
- **Business Environment**
  - Data fragmentation can be good for OLTP environment
  - Data fragmentation can be bad for Analysis Services environment

Depending on your business environment, fragmentation can be either good or bad for performance.

How Fragmentation Occurs

Fragmentation occurs when data is modified. For example, when rows of data are added to or deleted from a table, or when values in the indexed columns are changed, SQL Server adjusts the index pages to accommodate the changes and to maintain the storage of the indexed data. The adjustment of the index pages is known as a *page split*. The splitting process increases the size of a table and the time that is needed to process queries.

Methods of Managing Fragmentation

There are two methods of managing fragmentation in SQL Server. The first method is to drop and recreate a clustered index and to specify a fillfactor value by using the FILLFACTOR option. The second method is to rebuild an index and specify a fillfactor value.

Business Environment

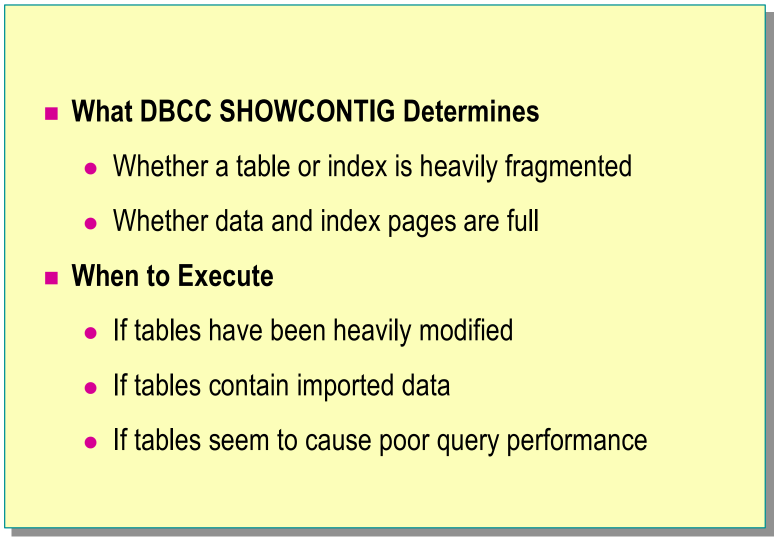
The degree of fragmentation that is acceptable in your database depends on your environment:

- In an OLTP environment, fragmentation can be beneficial, because an OLTP environment is write-intensive. A typical OLTP system has large numbers of concurrent users who are actively adding and modifying data.
- Fragmentation can be detrimental in an Analysis Services environment because that environment is read-intensive.

DBCC SHOWCONTIG Statement

Topic Objective
To explain the use of the DBCC SHOWCONTIG statement.

Lead-in
If you're concerned about fragmentation, use the DBCC SHOWCONTIG statement to determine whether your table or indexes are fragmented.



- **What DBCC SHOWCONTIG Determines**
  - Whether a table or index is heavily fragmented
  - Whether data and index pages are full
- **When to Execute**
  - If tables have been heavily modified
  - If tables contain imported data
  - If tables seem to cause poor query performance

Delivery Tip
Demonstrate DBCC SHOWCONTIG statement on a table in the **ClassNorthwind** database and do a high-level review of the output.

The DBCC SHOWCONTIG statement displays fragmentation information on the data and indexes of a specified table.

What DBCC SHOWCONTIG Statement Determines

When you execute the DBCC SHOWCONTIG statement, SQL Server goes across the index pages at the leaf level to determine whether a table or specified index is heavily fragmented. The DBCC SHOWCONTIG statement also determines whether the data and index pages are full.

When to Execute

Execute the DBCC SHOWCONTIG statement on heavily modified tables, tables that contain imported data, or tables that seem to cause poor query performance. When you execute the DBCC SHOWCONTIG statement, consider the following facts and guidelines:

Delivery Tip
Demonstrate one of the two ways to obtain the table ID of a table. One way uses the OBJECT_ID function. The other way involves querying the **sysindexes** system table of the **ClassNorthwind** database.

- SQL Server requires you to reference either a table or index ID when you execute the DBCC SHOWCONTIG statement. Query the **sysindexes** table to obtain the table or index ID.
- Determine how often you should execute the DBCC SHOWCONTIG statement. Measure the activity level on a table on a daily, weekly, or monthly basis.

The following table describes the statistics that the DBCC SHOWCONTIG statement returns.

Statistic	Description
Pages scanned	Number of pages in the table or index.
Extents scanned	Number of extents in the table or index.
Extent switches	Number of times that the DBCC statement left an extent while it was traversing the pages of the extent.
Average pages per extent	Number of pages per extent in the page chain.
Scan density [Best Count: Actual Count]	The number in scan density is 100 (a percentage) if everything is contiguous; if it is below 100, some fragmentation exists. Best Count is the ideal number of extent changes that would be present if everything were contiguously linked. Actual Count is the actual number of extent changes.
Logical scan fragmentation	Percentage of out-of-order pages returned from scanning the leaf pages of an index. This number is not relevant to heaps and text indexes. An out-of-order page is one for which the next page indicated in an Index Allocation Map (IAM) is a different page than the page pointed to by the next-page pointer in the leaf page.
Extent scan fragmentation	Percentage of out-of-order extents in scanning the leaf pages of an index. This number is not relevant to heaps. An out-of-order extent is one for which the extent containing the current page for an index is not the next physical extent—after the extent containing the previous page for an index.
Average bytes free per page	Average number of free bytes on the scanned pages. The higher the number, the less full the pages are—lower numbers are better. Be aware, however, that this number is also affected by row size. A large row size may result in a higher number.
Average page density (full)	Value that shows the fullness of a page. This value considers row size, so it is a more accurate indication of the fullness of a page. Higher percentages are better than lower percentages.

Syntax

```
DBCC SHOWCONTIG
  [({table_name | table_id | view_name | view_id }
  [, index_name | index_id ] )]
  [ WITH
  { ALL_INDEXES | FAST
  [, ALL_INDEXES ] | TABLERESULTS
  [, { ALL_INDEXES } ]
  [, { FAST | ALL_LEVELS } ]
          }
  ]
```

Example

This example executes a statement that accesses the **Customers** table.

```
USE Northwind
DBCC SHOWCONTIG (Customers, PK_Customers)
```

Result

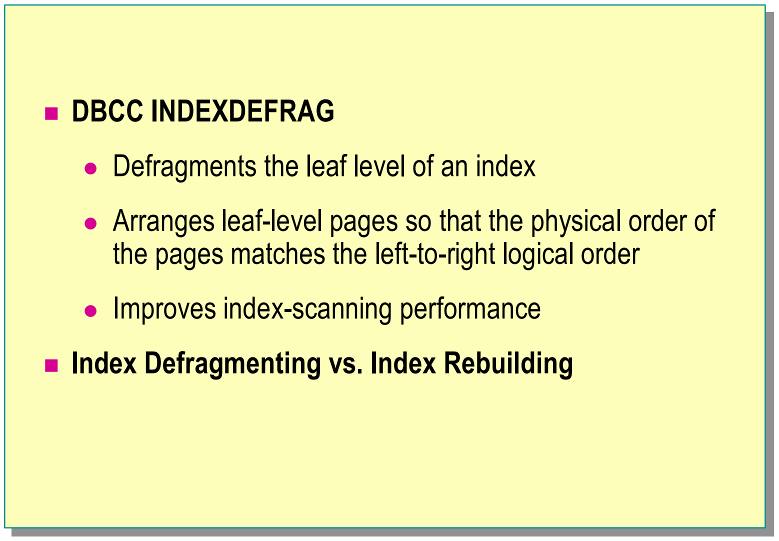
```
DBCC SHOWCONTIG scanning 'Customers' table...
Table: 'Customers' (2073058421); index ID: 1, database ID: 6
TABLE level scan performed.
Pages Scanned:                          3
Extents Scanned:                        2
Extent Switches:                        1
Avg. Pages per Extent:                  1.5
Scan Density [Best Count:Actual Count]:   50.00% [1:2]
Logical Scan Fragmentation              0.00%
Extent Scan Fragmentation:              50.00%
Avg. Bytes Free per Page:               246.7
Avg. Page Density (full):               96.95%
DBCC execution completed. If DBCC printed error messages,
contact your system administrator.
```

Delivery Tip
Compare whether these results are appropriate in a given environment.

DBCC INDEXDEFRAG Statement

Topic Objective
To explain the use of the DBCC INDEXDEFRAG statement.

Lead-in
The DBCC INDEXDEFRAG statement can defragment the leaf level of clustered and nonclustered indexes on tables and views.



- **DBCC INDEXDEFRAG**
  - Defragments the leaf level of an index
  - Arranges leaf-level pages so that the physical order of the pages matches the left-to-right logical order
  - Improves index-scanning performance
- **Index Defragmenting vs. Index Rebuilding**

As data in a table changes, the indexes on the table sometimes become *fragmented*. The DBCC INDEXDEFRAG statement can *defragment* the leaf level of clustered and nonclustered indexes on tables and views. Defragmenting arranges the pages so that the physical order of the pages matches the left-to-right logical order of the leaf nodes. This rearrangement improves index-scanning performance.

Using the DBCC INDEXDEFRAG Statement

When you use DBCC INDEXDEFRAG, it:

- Compacts the pages of an index, taking into account the FILLFACTOR specified when the index was created. Any empty pages created as a result of this compaction will be removed.
- Defragments one file at a time when an index spans more than one file. Pages do not migrate between files.
- Reports to the user an estimated percentage completed. Reporting is done every five minutes. The DBCC INDEXDEFRAG statement can be terminated at any point in the process, and any completed work is retained.
- Is an online operation. It does not hold locks for an extended time, and does not block running queries or updates. Defragmentation is always fully logged, regardless of the database recovery model setting.

Index Defragmenting vs. Index Rebuilding

The time required to defragment is related to the amount of fragmentation. A very fragmented index might require more time to defragment than to rebuild. A relatively unfragmented index defragments faster than rebuilding a new index.

Note Using the DBCC INDEXDEFRAG statement does not improve performance when indexes are physically defragmented on disk. To physically defragment an index, rebuild the index.

Syntax

```
DBCC INDEXDEFRAG
    ( { database_name | database_id | 0 }
        , { table_name | table_id | 'view_name' | view_id }
        , { index_name | index_id }
    )   [ WITH NO_INFOMSGS ]
```

Example

This example executes the DBCC INDEXDEFRAG statement on the **mem_no_CL** index of the **Member** table in the **credit** database.

```
DBCC INDEXDEFRAG(credit, member, mem_no_CL)
```

Result

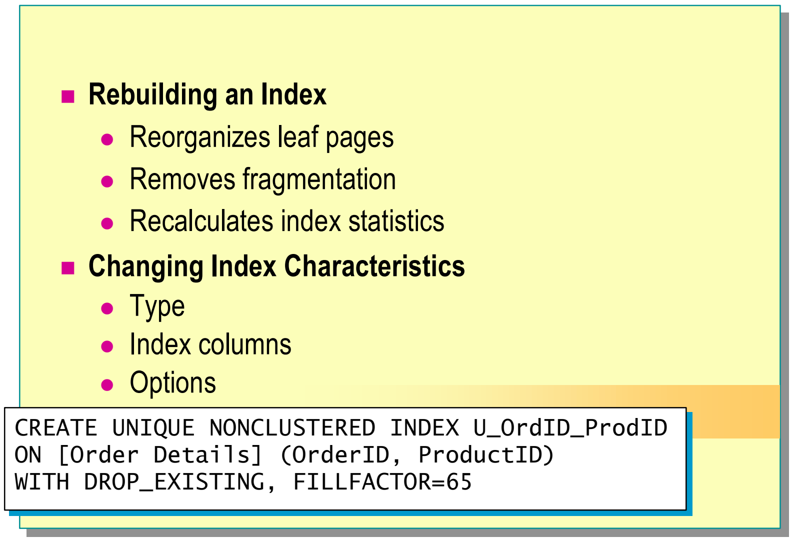
```
Pages scanned        Pages moved           Pages removed
150                  28                    9

(1 row(s) affected)
```

DROP_EXISTING Option

Topic Objective
To discuss using the DROP_EXISTING option to change the index definition or rebuild indexes.

Lead-in
Use the DROP_EXISTING option to change the index definition or to rebuild indexes.



- **Rebuilding an Index**
  - Reorganizes leaf pages
  - Removes fragmentation
  - Recalculates index statistics
- **Changing Index Characteristics**
  - Type
  - Index columns
  - Options

```
CREATE UNIQUE NONCLUSTERED INDEX U_OrdID_ProdID
ON [Order Details] (OrderID, ProductID)
WITH DROP_EXISTING, FILLFACTOR=65
```

Use the DROP_EXISTING option to change the characteristics of an index or to rebuild indexes without having to drop the index and recreate it. The benefit of using the DROP_EXISTING option is that you can modify indexes created with PRIMARY KEY or UNIQUE constraints.

Rebuilding an Index

Execute the CREATE INDEX statement with the DROP_EXISTING option to rebuild a named clustered or nonclustered index:

- Reorganize the leaf-level pages by compressing or expanding rows
- Remove fragmentation
- Recalculate the index statistics

For Your Information
Do not teach the DBCC DBREINDEX statement. DROP_EXISTING provides more functionality. The only advantage of DBCC DBREINDEX is that you can rebuild multiple indexes simultaneously.

Changing Index Characteristics

When you use the DROP_EXISTING option, you can change the following index characteristics:

- Type
  - You can change a nonclustered index into a clustered index.
  - You cannot change a clustered index into a nonclustered index.
- Index columns
  - You can change the index definition to specify different columns.
  - You can specify additional columns or remove specified columns from a composite index.
  - You can change the index columns to be unique or not unique.
- Options
  - You can change the FILLFACTOR or PAD INDEX percentage value.

When you use the DROP_EXISTING option, consider the following facts and guidelines:

- For a clustered index, SQL Server requires that you have 1.2 times the amount of table space to physically reorganize the data.
- The DROP_EXISTING option accelerates the process of building clustered and nonclustered indexes by eliminating the sorting process.
- Use the FILLFACTOR option with the DROP_EXISTING option if you want your leaf-level pages to fill to a certain percentage.

  This can be useful if space must be allocated for new data or if the index must be compacted.
- You cannot rebuild indexes on system tables.
- The DROP_EXISTING option on a clustered index helps you avoid the unnecessary work of deleting and re-creating nonclustered indexes if the clustered index is rebuilt on the same column.
- The nonclustered indexes are rebuilt once, and only if the keys are different.

Example

This example rebuilds the existing index, **U_OrdID_ProdID** , for the **Order Details** table. The index is redefined as a clustered, composite index with a specified option of filling each data page to 65 percent. This statement will fail if a clustered index already exists on the **Order Details** table.

```
CREATE UNIQUE NONCLUSTERED INDEX U_OrdID_ProdID
ON [Order Details] (OrderID, ProductID)
WITH DROP_EXISTING, FILLFACTOR=65
```

Lab A: Creating and Maintaining Indexes

Topic Objective
To introduce the lab.

Lead-in
In this lab, you will create indexes and determine the size and density of indexes.

Explain the lab objectives.

Objectives

After completing this lab, you will be able to:

- Create indexes.
- Determine the size and density of indexes.

Prerequisites

Before working on this lab, you must have:

- Script files for this lab, which are located in C:\Moc\2073A\Labfiles\L07.
- Answer files for this lab, which are located in C:\Moc\2073A\Labfiles\L07\Answers.

Lab Setup

To complete this lab, you must have either:

- Completed the prior lab, or
- Executed the C:\Moc\2073A\Batches\Restore07A.cmd batch file.

  This command file restores the **ClassNorthwind** database to a state required for this lab.

For More Information

If you require help with executing files, search SQL Query Analyzer Help for “Execute a query”.

Other resources that you can use include:

- The **Northwind** database schema.
- The **credit** database schema.
- Microsoft SQL Server Books Online.

Scenario

The organization of the classroom is meant to simulate that of a worldwide trading firm named Northwind Traders. Its fictitious domain name is nwtraders.msft. The primary DNS server for nwtraders.msft is the instructor computer, which has an Internet Protocol (IP) address of 192.168.*x*.200 (where *x* is the assigned classroom number). The name of the instructor computer is London.

The following table provides the user name, computer name, and IP address for each student computer in the fictitious **nwtraders.msft** domain. Find the user name for your computer, and make a note of it.

User name	Computer name	IP address
SQLAdmin1	Vancouver	192.168.*x*.1
SQLAdmin2	Denver	192.168.*x*.2
SQLAdmin3	Perth	192.168.*x*.3
SQLAdmin4	Brisbane	192.168.*x*.4
SQLAdmin5	Lisbon	192.168.*x*.5
SQLAdmin6	Bonn	192.168.*x*.6
SQLAdmin7	Lima	192.168.*x*.7
SQLAdmin8	Santiago	192.168.*x*.8
SQLAdmin9	Bangalore	192.168.*x*.9
SQLAdmin10	Singapore	192.168.*x*.10
SQLAdmin11	Casablanca	192.168.*x*.11
SQLAdmin12	Tunis	192.168.*x*.12
SQLAdmin13	Acapulco	192.168.*x*.13
SQLAdmin14	Miami	192.168.*x*.14
SQLAdmin15	Auckland	192.168.*x*.15
SQLAdmin16	Suva	192.168.*x*.16
SQLAdmin17	Stockholm	192.168.*x*.17
SQLAdmin18	Moscow	192.168.*x*.18
SQLAdmin19	Caracas	192.168.*x*.19
SQLAdmin20	Montevideo	192.168.*x*.20
SQLAdmin21	Manila	192.168.*x*.21
SQLAdmin22	Tokyo	192.168.*x*.22
SQLAdmin23	Khartoum	192.168.*x*.23
SQLAdmin24	Nairobi	192.168.*x*.24

Estimated time to complete this lab: 30 minutes

Exercise 1
Creating Indexes

In this exercise, you will create several indexes to complement FOREIGN KEY constraints on tables in the **ClassNorthwind** database.

► **To create an index on the Orders table**

In this procedure, you will open a script file that creates an index, review the contents of the script, execute it, and then verify that the index was created.

1. Log on to the **NWTraders** classroom domain by using the information in the following table.

Option	Value
User name	**SQLAdmin*x*** (where *x* corresponds to your computer name as designated in the **nwtraders.msft** classroom domain)
Password	**password**

2. Open SQL Query Analyzer and, if requested, log in to the (local) server with Microsoft Windows Authentication.

   You have permission to log in to and administer SQL Server because you are logged as **SQLAdmin*x***, which is a member of the Windows 2000 local group, Administrators. All members of this group are automatically mapped to the SQL Server **sysadmin** role.

3. In the **DB** list, click **ClassNorthwind**.
4. Open the C:\Moc\2073A\Labfiles\L07\CreaIndx1.sql script file.
5. Review the CREATE INDEX statement.

   This script creates a nonclustered index named **Orders_Customers_link** on the **CustomerID** column in the **Orders** table with a fillfactor value of 75.

6. Execute the script file.
7. Verify that the **Orders_Customers_link** index was created by executing the following statement:

```
EXEC sp_help Orders
```

   The results of the **sp_help** system stored procedure show that the **Orders_Customers_link** nonclustered index exists on the **CustomerID** column in the **Orders** table.

► **To create indexes on foreign keys that reference the Products table**

In this procedure, you will create clustered and nonclustered indexes for all foreign key references in the **Products** table by using the following information. You can use the Create Index wizard in SQL Server Enterprise Manager or write a Transact-SQL statement in SQL Query Analyzer. C:\Moc\2073A\Labfiles\L07\Answers\CreaIndx2.sql is a completed script for this procedure.

1. Verify that you are using the **ClassNorthwind** database.
2. Write and execute a script that creates the following indexes.

Index type	Name	Table	Column	Fillfactor value
Clustered	**Products_CategoryID_link**	**Products**	**CategoryID**	0
Nonclustered	**Products_SupplierID_link**	**Products**	**SupplierID**	0

```
CREATE    CLUSTERED INDEX Products_CategoryID_link ON
     Products(CategoryID)

CREATE NONCLUSTERED INDEX Products_SupplierID_link ON
     Products(SupplierID)
```

3. Query the **sysindexes** system table to verify that the indexes were created.

► **To verify the existence of the indexes that you created**

In this procedure, you will execute statements to verify that the indexes that you created exist and are correct.

1. Execute the **sp_helpindex** system stored procedure on the **Orders** table.

   What are the results?

   PK_Orders, Orders_Customers_link.

   ____

   ____

2. Execute the **sp_helpindex** system stored procedure on the **Products** table. Why are there indexes on the foreign key columns?

   An index on a foreign key is not required but is recommended to associate foreign key members with the primary key.

   ____

   ____

   Why are all of the indexes not unique?

   Because these indexes represent foreign keys. Often the relationship between primary and foreign keys is a one-to-many relationship. Therefore, one unique key value in the primary key can have relationships with many of the same key values in the foreign key column.

   ____

   ____

Exercise 2
Examining Index Structures

In this exercise, you will use SQL Query Analyzer to examine the table structure before creating indexes. You will create various types of indexes with different fillfactors and observe the effects on the table structure.

You can open, review, and execute sections of the ExamIndex.sql script file in the C:\Moc\SQL2073A\Labfiles\L07 folder, or type and execute the provided Transact-SQL statements.

► **To observe the initial table structure**

In this procedure, you will execute a Transact-SQL statement to obtain information about the **Member** table.

1. Type and execute these statements individually to obtain information about the **Member** table:

```
USE credit
GO
EXEC sp_spaceused member

SELECT * FROM sysindexes WHERE id = OBJECT_ID('member')

DBCC SHOWCONTIG ('member')
```

2. Record the statistical information in the following table.

Information	Source	Result
Number of rows	**sp_spaceused**: rows	**10,000**
Number of indexes	**SELECT * FROM sysindexes WHERE id = OBJECT_ID('member')**	**None. A row in sysindexes with a 0 for the value of *indid* represents the table itself.**
Number of pages	SHOWCONTIG: Pages Scanned	**150**
Number of rows per page	Calculate and round up the results. (# of rows/ # of pages) = # of rows per page	**67**
Number of extents	SHOWCONTIG: Extent Switches	**18**
Average extent fill	SHOWCONTIG: Avg. Pages per Extent	**8**
Average page fill	SHOWCONTIG: Avg. Page Density (full)	**98%**

► To create a clustered index

In this procedure, you will create a unique clustered index and observe changes to the table structure. You also will obtain information about the index structure.

1. Type and execute this statement to create a unique clustered index on the **member_no** column of the **Member** table, without specifying a fillfactor:

```
USE credit
CREATE UNIQUE CLUSTERED INDEX mem_no_CL
   ON member (member_no)
```

2. Type and execute the following statement to obtain information about the **Member** table:

```
USE credit
SELECT * FROM sysindexes WHERE id = OBJECT_ID('member')

DBCC SHOWCONTIG ('member')
```

3. Record the statistical information in the following table.

Information	**Source**	**Result**
Number of clustered index pages	**sysindexes** row: **used**	**145**
Number of data pages in the clustered index	**sysindexes** row: **dpages**	**142**
Number of non-data pages in the clustered index	(**used** – **dpages**)	**145 – 142 = 3**
Number of indexes	**SELECT * FROM sysindexes**	**One. A row in sysindexes with a 1 for the value of *indid* represents the clustered index.**
Number of pages	SHOWCONTIG: Pages Scanned	**142**
Number of rows per page	Calculate and round up the results. (# of rows/ # of pages) = # of rows per page	**71**
Number of extents	SHOWCONTIG: Extent Switches	**17**
Average extent fill	SHOWCONTIG: Avg. Pages per Extent	**8**
Average page fill	SHOWCONTIG: Avg. Page Density (full)	**99%**

Are the pages still full?

Yes.

Is the table still contiguous?

Yes.

Will creating a clustered index always make the data pages more compact? Why or why not?

No. Creating a clustered index may not always compact the pages. It depends on the nature of the data. Pages can be compacted if the table is fragmented because of updates to the key value or unordered insertions and deletions. If the pages are already compacted, creating a clustered index will have no effect.

► **To create a nonclustered index**

In this procedure, you will create a nonclustered index and obtain information about the index structure.

1. Type and execute this statement to drop the previously created index:

```
USE credit
EXEC index_cleanup member
```

2. Type and execute this statement to create a nonclustered index on the **firstname** column of the **Member** table, without specifying a fillfactor:

```
USE credit
CREATE NONCLUSTERED INDEX indx_fname
   ON member(firstname)
```

3. Type and execute this SELECT statement that returns the **sysindexes** rows for the **Member** table:

```
USE credit
SELECT * FROM sysindexes WHERE id = OBJECT_ID('member')
```

4. Record the statistical information in the following table.

Information	Source	Result
Number of pages in the nonclustered index on the **firstname** column	**sysindexes** row: used	**40**
Number of pages in the leaf level	**sysindexes** row: **dpages**	**37**
Approximate number of rows per leaf page	(# rows in table/# leaf-level pages)	**(10000 / 37) = 271**

► To create a nonclustered index with a fillfactor

In this procedure, you will create a nonclustered index and observe changes to the table structure.

1. Type and execute this statement to drop the nonclustered index from the **Member** table:

```
USE credit
EXEC index_cleanup member
```

2. Type and execute this statement to create the same index, with a fillfactor of 25 percent:

```
USE credit
CREATE NONCLUSTERED INDEX indx_fname
   ON member(firstname)
   WITH FILLFACTOR=25
```

3. Type and execute this SELECT statement that returns the **sysindexes** rows for the **Member** table:

```
USE credit
SELECT * FROM sysindexes WHERE id = OBJECT_ID('member')
```

4. Record the statistical information in the following table.

Information	Source	Result
Number of pages in this index	**sysindexes** row: used	**148**
Number of pages in the leaf level	**sysindexes** row: **dpages**	**145**
Approximate number of rows per leaf page	(# rows in table/# leaf-level pages)	**(10000 / 145) = 68**

Is the increase in the leaf-level size in proportion to the fillfactor?

Yes, the increase is proportional.

____

How can you determine whether the increase in the leaf-level size is proportional to the fillfactor of 25 percent?

Using a fillfactor of 0 (default), you can fit 271 rows per leaf-level page. Multiply 271 by 25 percent, and the result is 68 rows. This is the number of rows that will fit on a leaf-level page that is only 25 percent full.

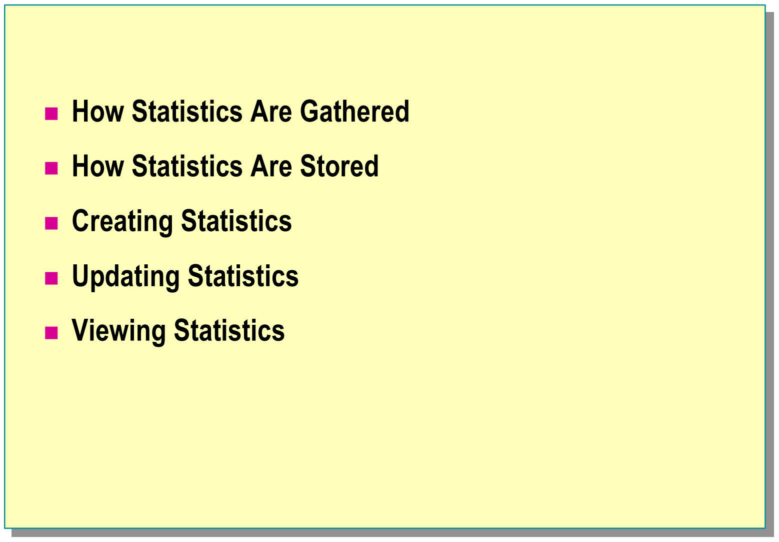
____

____

◆ Introduction to Statistics

Topic Objective
To introduce the topics in this section.

Lead-in
Statistics are created on indexes and can be created on columns. Because the query optimizer uses statistics to optimize queries, you should know how they are gathered, stored, created, updated, and viewed.



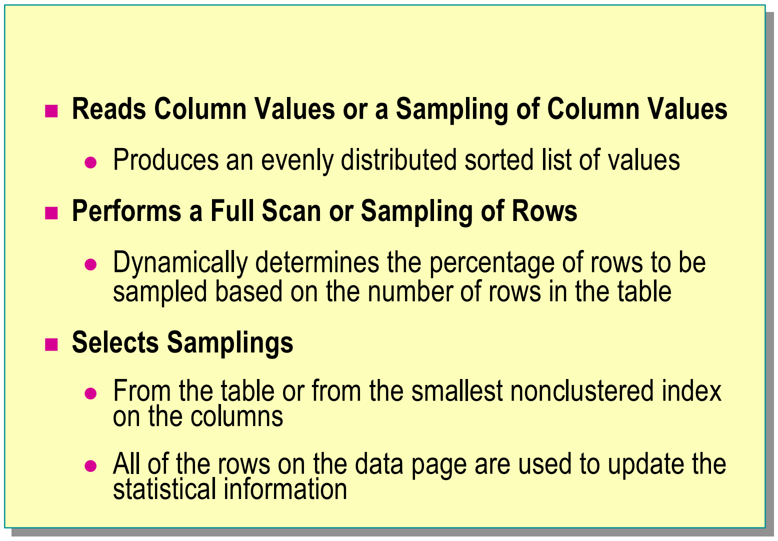
- **How Statistics Are Gathered**
- **How Statistics Are Stored**
- **Creating Statistics**
- **Updating Statistics**
- **Viewing Statistics**

Statistics are created on indexes and can be created on columns. Because the query optimizer uses statistics to optimize queries, you should know how they are gathered, stored, created, updated, and viewed.

How Statistics Are Gathered

Topic Objective
To discuss how SQL Server gathers statistics.

Lead-in
Statistics are a sampling of column values.



- **Reads Column Values or a Sampling of Column Values**
  - Produces an evenly distributed sorted list of values
- **Performs a Full Scan or Sampling of Rows**
  - Dynamically determines the percentage of rows to be sampled based on the number of rows in the table
- **Selects Samplings**
  - From the table or from the smallest nonclustered index on the columns
  - All of the rows on the data page are used to update the statistical information

Statistics are a sampling of column values.

Reads Column Values or a Sampling of Column Values

SQL Server gathers statistics by reading all of the column values or a sampling of column values to produce an evenly distributed and sorted list of values known as *distribution steps*. SQL Server generates distribution steps by performing a full scan or sample scan and then by selecting samplings.

Performs a Full Scan or Sampling of Rows

SQL Server dynamically determines the percentage of rows to be sampled based on the number of rows in the table. The query optimizer performs either a full scan or a sampling of rows when gathering statistics.

- The SAMPLE option is the default for updating and creating statistics.
- The FULLSCAN option is used when:
  - Indexes are created.
  - The FULLSCAN option is specified in the CREATE STATISTICS statement.
  - The UPDATE STATISTICS statement is executed.

Selects Samplings

The sampling is randomly selected across data pages from the table or from the smallest nonclustered index on the columns needed by the statistics. After a data page has been read from disk, all of the rows on the data page are used to update the statistical information.

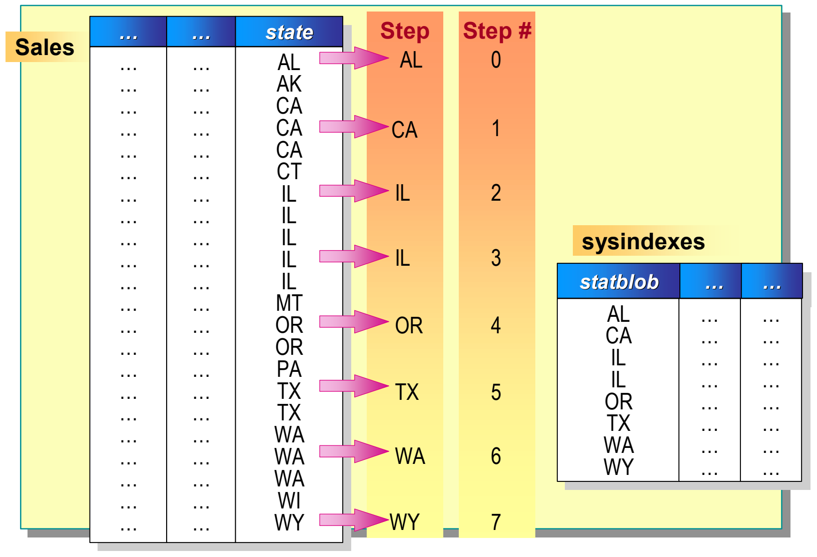
When the query optimizer gathers samplings:

- The table size determines which method is chosen.
- A minimum number of values are sampled to derive useful statistics.
- If the number of rows specified is too few to be useful, the query optimizer automatically corrects the sampling, based on the number of existing rows in the table.
- Statistics are kept only on the first column defined in a composite index.

How Statistics Are Stored

Topic Objective
To discuss how statistics are stored in SQL Server.

Lead-in
Statistics are stored in the **statblob** column of the **sysindexes** system table.



Delivery Tip
When you build an index on the **Sales** table, SQL Server automatically builds statistics.

This example shows the process of how SQL Server creates statistics.

Statistics are stored in the **statblob** column of the **sysindexes** system table.

Distribution Steps

Each value stored in the **statblob** column is called a distribution step. Distribution steps refer to space between data samplings, or how many rows are stepped across before the next sampling is taken and stored. The first and last key values in the index are always included in the statistics. There can be as many as 300 values, of which the end point is the 300th value.

Contents in the statblob Column

In addition to storing distribution steps, the **statblob** column also stores:

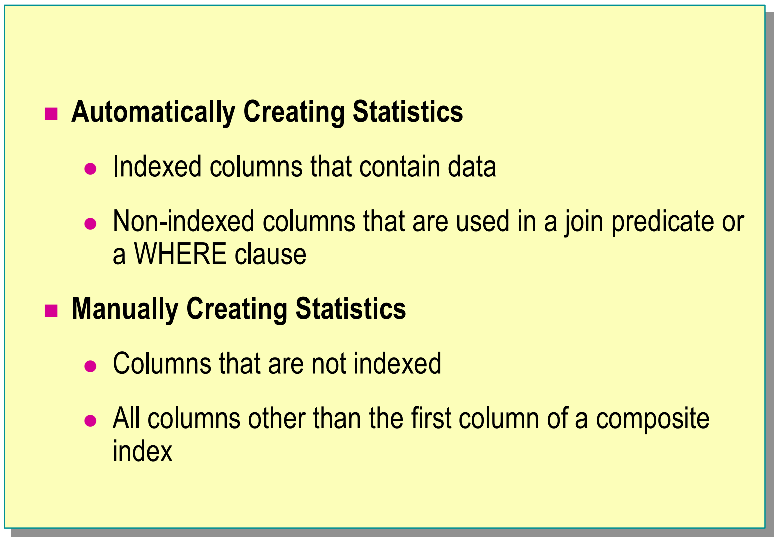
- Date and time when statistics were last updated.
- Number of rows in the table.
- Number of rows sampled to create the histogram and determine density.
- Number of distribution steps.
- Average key length.
- Density for individual columns and all of the columns combined.
- Number of rows that fall within a histogram step.
- Number of rows that are equal in value to the upper bound of the histogram step.
- Number of distinct values within a histogram step.

Note The **statblob** column is defined as an **image** data type.

Creating Statistics

Topic Objective
To discuss when statistics are created and when to create them.

Lead-in
You can create statistics automatically or manually. However, you should allow SQL Server to create statistics automatically for you.



- **Automatically Creating Statistics**
  - Indexed columns that contain data
  - Non-indexed columns that are used in a join predicate or a WHERE clause
- **Manually Creating Statistics**
  - Columns that are not indexed
  - All columns other than the first column of a composite index

You can create statistics automatically or manually. However, you should allow SQL Server to create statistics automatically for you.

Automatically Creating Statistics

When the **auto create statistics** database option default is set to ON, SQL Server automatically creates statistics for:

- Indexed columns that contain data.
- Non-indexed columns that are used in a join predicate or a WHERE clause.

The query optimizer activates the automatic creation of statistics when optimizing a query. This can be a disadvantage if the query optimizer determines that statistics are missing. The execution plan will include the statistics creation action, which requires additional time when processing the query.

Note When you execute a query and view the execution plan, the query optimizer may suggest remedial action, such as creating or updating statistics, or creating an index. At that point, you can immediately create or update statistics and indexes.

Manually Creating Statistics

You can execute the CREATE STATISTICS statement to create a histogram and associated density groups for specific columns. You can create statistics on:

- Columns that are not indexed.
- All columns other than the first column of a composite index.
- Computed columns only if the conditions are such that an index can be created on these columns.
- Columns that are not defined of **image**, **text**, and **ntext** data types.

Manually creating statistics is useful when you have a column that may not benefit from an index, but statistics on that column may be useful for creating more optimal execution plans. Having statistics on those columns eliminates the overhead of an index, while allowing the query optimizer to use the column when optimizing queries.

Note You must be the table owner to manually create statistics on a table.

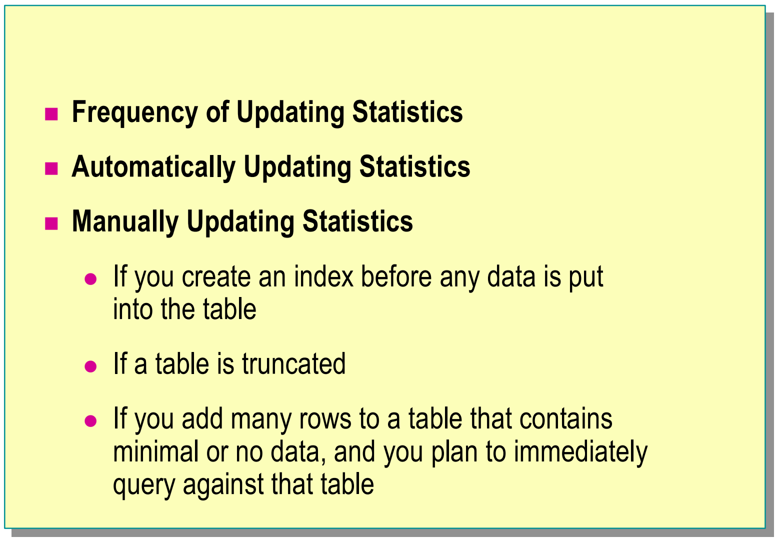
Partial Syntax

CREATE STATISTICS *statistics_name* ON {*table*| *view*} (*column* [,...*n*])

Updating Statistics

Topic Objective
To describe how SQL Server determines how often to update statistics, and when to update them.

Lead-in
Over time, statistics can become outdated, which can affect the performance of the query optimizer.



- **Frequency of Updating Statistics**
- **Automatically Updating Statistics**
- **Manually Updating Statistics**
  - If you create an index before any data is put into the table
  - If a table is truncated
  - If you add many rows to a table that contains minimal or no data, and you plan to immediately query against that table

Over time, statistics can become outdated, which can affect the performance of the query optimizer.

Frequency of Updating Statistics

SQL Server updates statistical information when the information becomes outdated. The volume of data in the column relative to the amount of changing data determines the frequency of the update. For example:

- The statistics for a table containing 10,000 rows may require updating when 1,000 index values have changed, because 1,000 index values represent a significant percentage of the table.
- The statistics for a table containing 10 million index entries may not require updating when 1,000 index values have changed, because 1,000 index values represents a small percentage of the table.

SQL Server always samples a minimum number of rows. Tables that are smaller than 8 megabytes (MB) are always fully scanned to gather statistics.

Note SQL Server issues a warning when statistics are out-of-date or unavailable. This warning appears when the execution plan is viewed by using the execution plan. You can use SQL Profiler to monitor the **Missing Column Statistics** event class. This event class indicates when statistics are missing.

Automatically Updating Statistics

You should allow SQL Server to automatically update statistics for you. When the **auto update statistics** database option is set to ON (default), SQL Server automatically updates existing statistics when they become outdated.

For example, if a table is substantially updated since the last time that the statistics were created or updated, SQL Server automatically updates the statistics to optimize a query that uses the table.

The query optimizer activates the automatic updating of statistics when optimizing a query. This can be a disadvantage if the query optimizer determines that statistics are out-of-date. The execution plan will include the statistics update action, which requires additional time in processing the query.

Manually Updating Statistics

You can execute the UPDATE STATISTICS statement to update information about the distribution of key values for one or more statistics in a specified table. You may want to manually update statistics for a table or column in the following situations:

- If you create an index before any data is put into a table.
- If a table is truncated.
- If you add many rows to a table that contains minimal or no data, and you plan to immediately query against that table.

Partial Syntax

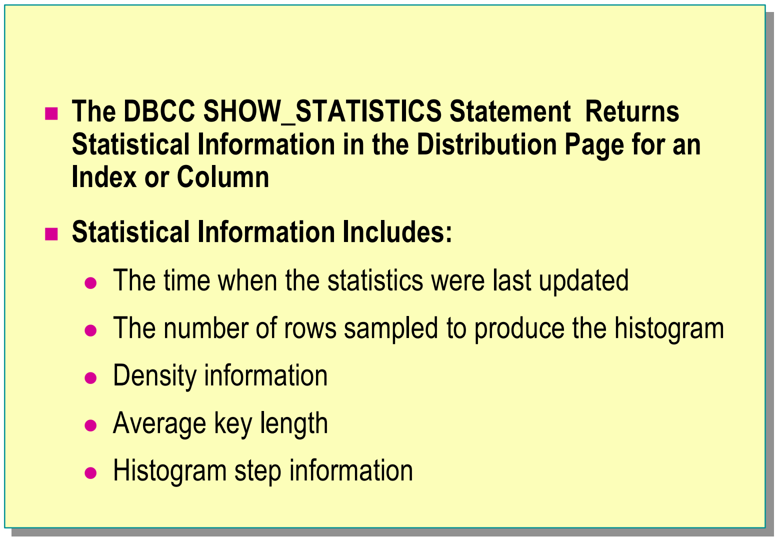
UPDATE STATISTICS *table*| *view* [*index* | (*statistics_name*[,...*n*])]

Note To see a list of index names and descriptions, execute the **sp_helpindex** system stored procedure with the table name.

Viewing Statistics

Topic Objective
To discuss how to view information about statistics.

Lead-in
You can view statistical information in the distribution page for an index or a column by executing the DBCC SHOW_STATISTICS statement.



- **The DBCC SHOW_STATISTICS Statement Returns Statistical Information in the Distribution Page for an Index or Column**
- **Statistical Information Includes:**
  - The time when the statistics were last updated
  - The number of rows sampled to produce the histogram
  - Density information
  - Average key length
  - Histogram step information

You can view statistical information in the distribution page for an index or a column by executing the DBCC SHOW_STATISTICS statement.

The following table describes the information that the DBCC SHOW_STATISTICS statement returns.

Column name	Description
Updated	Date and time the statistics were last updated
Rows	Number of rows in the table
Rows sampled	Number of rows sampled for statistics information
Steps	Number of distribution steps
Density	Selectivity of the first index column prefix (non-frequent)
Average key length	Average length of the first index column prefix
All density	Selectivity of a set of index column prefixes (frequent)
Average length	Average length of a set of index column prefixes
Columns	Names of index column prefixes for which **All density** and **Average length** are displayed
RANGE_HI_KEY	Upper bound value of a histogram step
RANGE_ROWS	Number of rows from the sample that fall in a histogram step, excluding the upper bound

(*continued*)

Column name	Description
EQ_ROWS	Number of rows from the sample that are equal in value to the upper bound of the histogram step
DISTINCT_RANGE_ROWS	Number of distinct values within a histogram step, excluding the upper bound
AVG_RANGE_ROWS	Average number of duplicate values within a histogram step, excluding the upper bound (RANGE_ROWS / DISTINCT_RANGE_ROWS for DISTINCT_RANGE_ROWS > 0)

Syntax

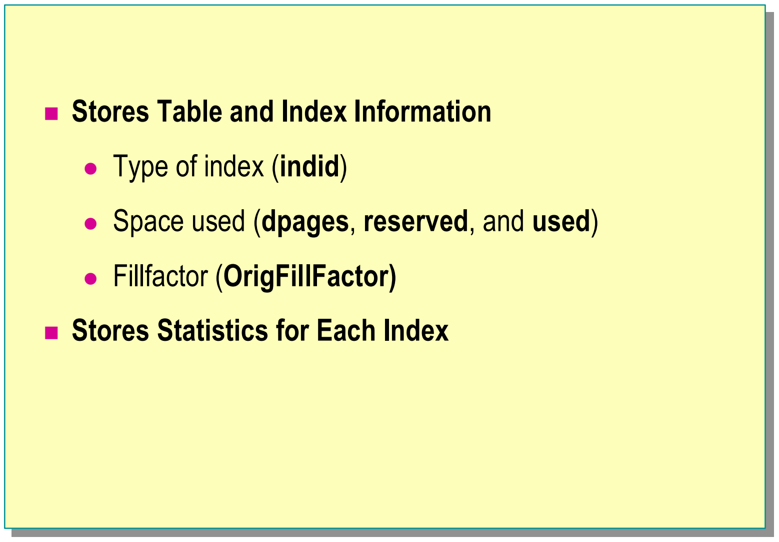
DBCC SHOW_STATISTICS (*table*, *target*)

Viewing statistics is typically useful when you do high-end performance tuning for specific queries. In most applications, it is not necessary to view statistics.

Querying the sysindexes Table

Topic Objective
To discuss querying the **sysindexes** table to get table and index information.

Lead-in
You can query the **sysindexes** table to get index and table information, in addition to statistics for each index.



- **Stores Table and Index Information**
  - Type of index (**indid**)
  - Space used (**dpages**, **reserved**, and **used**)
  - Fillfactor (**OrigFillFactor)**
- **Stores Statistics for Each Index**

You can query the **sysindexes** table to get index and table information, in addition to statistics for each index. The following table is a partial list of the information that you can view that comes from the data stored in the **sysindexes** table.

Column	Description	Values
indid (type of index)	ID of the index (type of index)	Possible values are: • 0 for nonclustered table • 1 for clustered index • >1 for nonclustered index • 255 for tables that have text or image data
dpages (space used)	Count of leaf-level index pages	For **indid** = 0 or **indid** = 1, **dpages** is the count of data pages used. For **indid**=255, **dpages** is set to 0. Otherwise, **dpages** is the count of nonclustered index pages used.
reserved (space used)	Count of reserved pages for an index	For **indid** = 0 or **indid** = 1, **reserved** is the count of pages allocated for all indexes and table data. For **indid** = 255, **reserved** is a count of the pages allocated for text or image data. Otherwise, **reserved** is the count of pages allocated for the index.
used (space used)	Count of space used by an index	For **indid** = 0 or **indid** = 1, **used** is the count of the total pages used for all index and table data. For **indid** = 255, **used** is a count of the pages used for text or image data. Otherwise, **used** is the count of pages used for the index.

(*continued*)

Column	Description	Values
OrigFillFactor (fillfactor)	Original fillfactor value used when the index was created	This value is not maintained; however, it can be helpful if you need to recreate an index and do not remember which fillfactor value was used.
minlen	Minimum size of a row	Integer value.
xmaxlen	Maximum size of a row	Integer value.
maxirow	Maximum size of a non-leaf index row	Integer value.
keys	Description of key columns	Applies only if entry is an index.
statversion	Number of times the statistics have been updated	Integer value.
statblob	Statistics binary large object (BLOB)	Stores statistical information.

Example

This example executes a statement that accesses the index ID and other information from the **sysindexes** system table. Specify the clustered index name (*index_name*) in the WHERE clause to obtain the index ID of a clustered index.

```
SELECT id, indid, reserved, used, origfillfactor, name
FROM Northwind.dbo.sysindexes
WHERE name = 'PK_customers'
```

Result

```
id            indid   reserved        used     origfillfactor    name
2073058421    1       15              15       0                 PK_Customers

(1 row(s) affected)
```

Setting Up Indexes Using the Index Tuning Wizard

Topic Objective
To introduce the Index Tuning Wizard.

Lead-in
Whether you are a novice or an advanced SQL Server user, the Index Tuning Wizard can help you create appropriate indexes on a new database or verify existing indexing on your current database.

- **Use the Index Tuning Wizard to:**
  - Recommend or verify optimal index configuration
  - Provide cost analysis reports
  - Recommend ways to tune the database
  - Specify criteria when a workload is evaluated
- **Do Not Use the Index Tuning Wizard on:**
  - Tables referenced by cross-database queries that do not exist
  - System tables, PRIMARY KEY constraints, unique indexes

Whether you are a novice or an advanced SQL Server user, the Index Tuning Wizard can help you create appropriate indexes on a new database or verify existing indexing on your current database. The Index Tuning Wizard looks at the query load to determine which indexes are useful, whereas the execution plan feature displays which indexes are used in queries.

Determining When to Use the Index Tuning Wizard

Novice users can use the wizard to quickly create an optimal index configuration. Advanced users can use the wizard for establishing a baseline index configuration. Advanced users can then custom-tune or verify their existing index configurations.

The Index Tuning Wizard can:

- Recommend or verify the optimal index configuration for a database, given an applied workload or trace file, by using the query optimizer costing analysis.
- Provide cost-analysis reports on the effects of the proposed changes, including:
  - Index usage on current and recommended indexes.
  - Query performance improvement for the 100 most expensive queries and table participation in a workload.
- Recommend ways to tune the database for a small set of problem queries.
- Specify criteria to consider when the Index Tuning Wizard evaluates a workload, such as maximum queries to tune, maximum space for recommended indexes, and maximum columns per index.

Determining How To Use the Wizard

When you want to use the Index Tuning Wizard, consider the following facts and guidelines:

- The user invoking the Index Tuning Wizard must be a member of the **sysadmin** fixed server role because the queries in the workload are analyzed in the security context of the user.
- It is not recommended that you use the Index Tuning Wizard on:
  - Tables referenced by cross-database queries that do not exist in the currently selected database.
  - System tables.
  - PRIMARY KEY constraints and unique indexes.

    The wizard may drop or replace a clustered index that is not unique, or currently created on a PRIMARY KEY constraint.
- It is not recommended that you drop any indexes when the **Keep all existing indexes** option is selected.

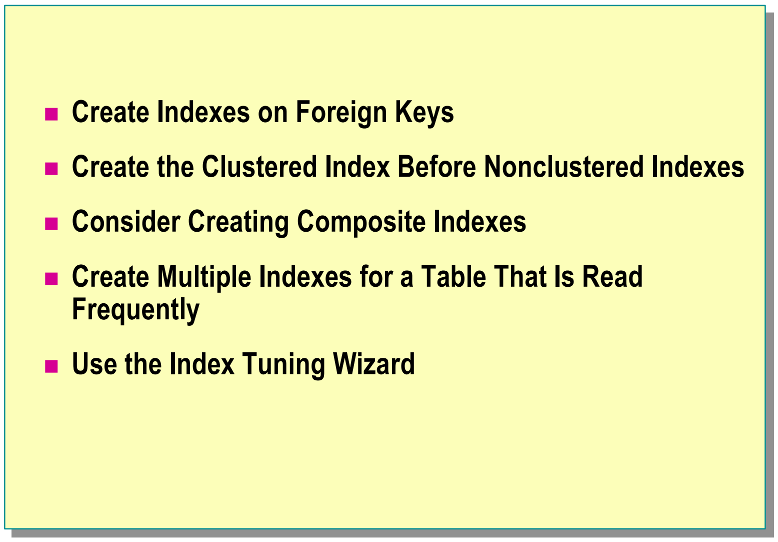
  The wizard recommends only new indexes, if appropriate. Clearing this option can result in a greater overall improvement in the performance of the workload.
- It is recommended that you leave the **Add indexed views** option selected.
- Hints can prevent the Index Tuning Wizard from choosing a better execution plan. Consider removing any index hints from queries before analyzing the workload.
- When you want to reduce the execution time of the Index Tuning Wizard, you should:
  - Ensure that **Perform thorough analysis** is not selected in the **Select Server and Database** dialog box. Performing a thorough analysis causes the Index Tuning Wizard to perform an exhaustive analysis of the queries, resulting in a longer execution time. Selecting this option can result in a greater overall improvement in the performance of the tuned workload.
  - Tune only a subset of the tables in the database.
  - Reduce the size of the workload file.

Note When you use the Index Tuning Wizard to analyze a Transact-SQL script that does not have a file name extension of .sql, such as My_script.txt, and you open the file with **File Format** set to **Auto**, the wizard generates the error message **Not a valid File Format**. Set **File Format** to **ANSI SQL** or **UNICODE SQL** instead.

Performance Considerations

Topic Objective
To discuss the performance considerations for planning and creating indexes.

Lead-in
Take the following actions to reduce the impact on performance when you create or use indexes.



- **Create Indexes on Foreign Keys**
- **Create the Clustered Index Before Nonclustered Indexes**
- **Consider Creating Composite Indexes**
- **Create Multiple Indexes for a Table That Is Read Frequently**
- **Use the Index Tuning Wizard**

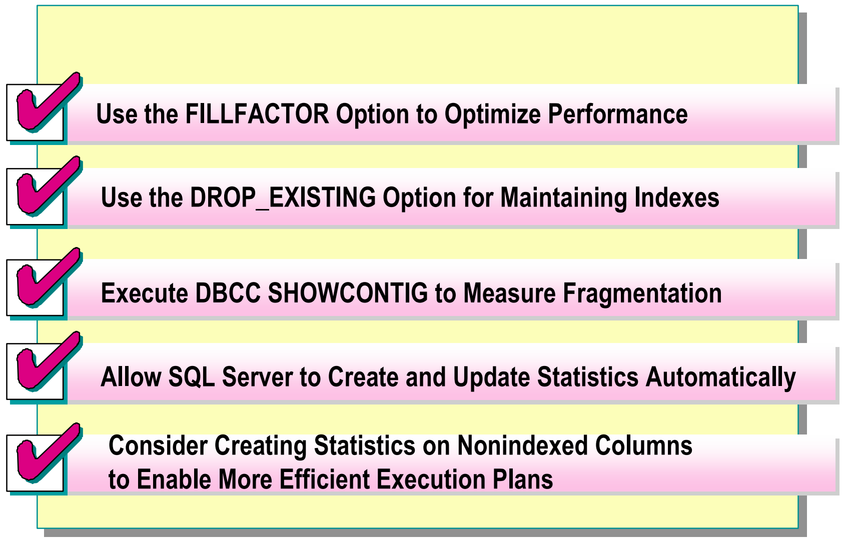
Take the following actions to reduce the impact on performance when you create or use indexes:

- Create indexes on foreign keys, because foreign keys are typically referenced in queries.
- Create the clustered index before nonclustered indexes, because a clustered index changes the physical row order of the table.
- Create composite indexes. Query performance is enhanced with composite indexes, especially when users regularly search for information in more than one way.
- Create multiple indexes for a table, especially if the table is read frequently. Query performance is enhanced when a table has a clustered index and nonclustered indexes.
- Use the Index Tuning Wizard to track the usage of your indexes automatically and to assist you with maintaining and creating indexes that perform optimally.

Recommended Practices

Topic Objective
To list the recommended practices for using or creating indexes.

Lead-in
To get the most out of your indexes, consider these recommended practices.



✓ **Use the FILLFACTOR Option to Optimize Performance**

✓ **Use the DROP_EXISTING Option for Maintaining Indexes**

✓ **Execute DBCC SHOWCONTIG to Measure Fragmentation**

✓ **Allow SQL Server to Create and Update Statistics Automatically**

✓ **Consider Creating Statistics on Nonindexed Columns to Enable More Efficient Execution Plans**

To get the most out of your indexes, consider the following practices:

- Use the FILLFACTOR option to optimize the performance of INSERT and UPDATE statements. This option allows you to specify a percentage of free space on the leaf-level pages.
- Use the DROP_EXISTING option to rebuild indexes quickly.
- Execute the DBCC SHOWCONTIG statement to determine the fragmentation of a table. The DBCC SHOWCONTIG statement shows the percentage of fragmentation and the average page density in a table.
- Allow SQL Server to create and update statistics for you automatically. Over time, statistics sometimes become outdated, which can affect the performance of the query optimizer. You should set **auto update statistics** and **auto create statistics** to ON.
- Consider creating statistics on nonindexed columns and secondary columns of a composite index. You can enhance query performance without incurring the overhead of maintaining additional indexes. Creating statistics allows the query optimizer to create more efficient execution plans.

Additional information on the following topics is available in SQL Server Books Online.

Topic	Search on
Index statistics	sp_autostats
	DBCC SHOW_STATISTICS
	"update statistics"
Creating indexes	"index tuning recommendations"
Computed columns	"SET options that affect results"
Functions	"deterministic and nondeterministic functions"

Lab B: Viewing Index Statistics

Topic Objective
To introduce the lab.

Lead-in
In this lab, you will execute queries and compare statistics.

Explain the lab objectives.

Objectives

After completing this lab, you will be able to:

- Estimate density and determine selectivity of indexes.
- View index statistics to determine whether the index is selective.

Prerequisites

Before working on this lab, you must have:

- Script files for this lab, which are located in C:\Moc\2073A\Labfiles\L07.
- Answer files for this lab, which are located in C:\Moc\2073A\Labfiles\L07\Answers.

Lab Setup

To complete this lab, you must have either:

- Completed the prior lab, or
- Executed the C:\Moc\2073A\Batches\Restore07B.cmd batch file.

  This command file restores the **ClassNorthwind** database to a state required for this lab.

For More Information

If you require help in executing files, search SQL Query Analyzer Help for "Execute a query".

Other resources that you can use include:

- The **credit** database schema.
- Microsoft SQL Server Books Online.

Scenario

The organization of the classroom is meant to simulate that of a worldwide trading firm named Northwind Traders. Its fictitious domain name is nwtraders.msft. The primary DNS server for nwtraders.msft is the instructor computer, which has an Internet Protocol (IP) address of 192.168.*x*.200 (where *x* is the assigned classroom number). The name of the instructor computer is London.

The following table provides the user name, computer name, and IP address for each student computer in the fictitious **nwtraders.msft** domain. Find the user name for your computer, and make a note of it.

User name	Computer name	IP address
SQLAdmin1	Vancouver	192.168.*x*.1
SQLAdmin2	Denver	192.168.*x*.2
SQLAdmin3	Perth	192.168.*x*.3
SQLAdmin4	Brisbane	192.168.*x*.4
SQLAdmin5	Lisbon	192.168.*x*.5
SQLAdmin6	Bonn	192.168.*x*.6
SQLAdmin7	Lima	192.168.*x*.7
SQLAdmin8	Santiago	192.168.*x*.8
SQLAdmin9	Bangalore	192.168.*x*.9
SQLAdmin10	Singapore	192.168.*x*.10
SQLAdmin11	Casablanca	192.168.*x*.11
SQLAdmin12	Tunis	192.168.*x*.12
SQLAdmin13	Acapulco	192.168.*x*.13
SQLAdmin14	Miami	192.168.*x*.14
SQLAdmin15	Auckland	192.168.*x*.15
SQLAdmin16	Suva	192.168.*x*.16
SQLAdmin17	Stockholm	192.168.*x*.17
SQLAdmin18	Moscow	192.168.*x*.18
SQLAdmin19	Caracas	192.168.*x*.19
SQLAdmin20	Montevideo	192.168.*x*.20
SQLAdmin21	Manila	192.168.*x*.21
SQLAdmin22	Tokyo	192.168.*x*.22
SQLAdmin23	Khartoum	192.168.*x*.23
SQLAdmin24	Nairobi	192.168.*x*.24

Estimated time to complete this lab: 30 minutes

Exercise 1
Examining the Use of Indexes

In this exercise, you will create indexes, execute a series of SELECT statements to examine the density of four columns in the **Charge** table, and determine selectivity.

You can open, review, and execute sections of the ExamUse.sql script file in the C:\Moc\SQL2073A\Labfiles\L07 folder, or type and execute the provided Transact-SQL statements.

► To create indexes

In this procedure, you will drop existing indexes on the **Charge** table and create nonclustered indexes.

1. Log on to the **NWTraders** classroom domain by using the information in the following table.

Option	Value
User name	**SQLAdmin*x*** (where *x* corresponds to your computer name as designated in the **nwtraders.msft** classroom domain)
Password	**password**

2. Open SQL Query Analyzer and, if requested, log in to the (local) server with Microsoft Windows® Authentication.

   You have permission to log in to and administer SQL Server because you are logged as **SQLAdmin*x***, which is a member of the Windows 2000 local group, Administrators. All members of this group are automatically mapped to the SQL Server **sysadmin** role.

3. Type and execute this statement to create unique, nonclustered indexes on the **charge_no**, **member_no**, **provider_no**, and **category_no** columns of the **charge** table:

```
USE credit
CREATE UNIQUE NONCLUSTERED INDEX charge_no_CL
          ON charge (charge_no)
CREATE NONCLUSTERED INDEX indx_member_no
          ON charge (member_no)
CREATE NONCLUSTERED INDEX indx_provider_no
          ON charge (provider_no)
CREATE NONCLUSTERED INDEX indx_category_no
          ON charge (category_no)
GO
```

► To review the charge table structure

In this procedure, you will creates indexes on the **Charge** table and determine the minimum and maximum values for the indexed columns.

1. Type or select these statements, and execute them to obtain the minimum and maximum values for the **charge_no**, **member_no**, **provider_no**, and **category_no** columns:

```
SELECT 'Charge_No ', MIN(Charge_No) AS 'Minimum Value',
      MAX(Charge_No) AS 'Maximum Value' FROM charge
UNION
SELECT 'Member_No ', MIN(Member_No) AS 'Minimum Value',
      MAX(Member_No) AS 'Maximum Value' FROM charge
UNION
SELECT 'Provider_No ', MIN(Provider_No) AS 'Minimum Value',
      MAX(Provider_No) AS 'Maximum Value' FROM charge
UNION
SELECT 'Category_No ', MIN(Category_No) AS 'Minimum Value',
      MAX(Category_No) AS 'Maximum Value' FROM charge
GO
```

2. Record the information in the following table.

Value	charge_no	member_no	provider_no	category_no
Min	**1**	**41**	**18**	**1**
Max	**100,000**	**10,000**	**500**	**10**

► To determine selectivity

In this procedure, you will execute a series of SELECT statements that select all rows from the **Charge** table. A table scan is performed for each SELECT statement. For each SELECT statement, you will first select the query and then view the estimated execution plan. You will modify the WHERE clause so that the query optimizer uses an index to retrieve the rows, and then you execute that query. After executing the query, you will record and evaluate the maximum number of rows that can be returned by using an index.

```
USE credit
SELECT * FROM charge
WHERE charge_no BETWEEN 1 AND 100000

USE credit
SELECT * FROM charge
WHERE member_no BETWEEN 1 AND 10000

USE credit
SELECT * FROM charge
WHERE provider_no BETWEEN 1 AND 500

USE credit
SELECT * FROM charge
WHERE category_no between 1 AND 10
```

1. Select the first statement, but do not execute it.
2. In the query window, on the **Query** menu, click **Display Estimated Execution Plan**.

   Notice the query plan for the statement.
3. Modify the range in the SELECT statements so that the query optimizer uses an index to retrieve the rows, rather than the optimizer using a table scan or full index scan.

   When choosing a range, remember that:

   - Each page has approximately 172 charges.
   - Older members and providers are less active than newer ones.
   - All categories are equally popular.
4. Execute the statements.
5. In the following table, record the maximum number of rows that can be returned by using an index.

WHERE clause	Approximate number of rows
WHERE **charge_no** BETWEEN 1 AND *n*	**116**
WHERE **member_no** BETWEEN 1 AND *n*	**222**
WHERE **provider_no** BETWEEN 1 AND *n*	**70**
WHERE **category_no** BETWEEN 1 AND *n*	**0**

6. Repeat steps 1 through 5 for the remaining SELECT statements.

Note You will notice that it is not easy to predict the selectivity of a query, even when you know the values of all of the arguments. It is best to let the query optimizer decide how to execute the query.

Is the number of rows accessed by the query optimizer the same for all indexes? Why or why not?

No. As selectivity decreases and distribution changes, indexes become less effective for the query optimizer to use because:

- **The charge numbers are evenly distributed throughout the table and they are unique. The predictability of the data is very accurate.**
- **The member number is less selective and is unevenly distributed. Estimating the number of rows that are returned becomes less accurate.**
- **The provider numbers are not evenly distributed in the charge table. The older providers (those with low provider numbers) do not have as many charges as newer providers. To obtain the same number of charges as newer providers, a larger percentage of the older providers is used when selecting charges.**
- **Categories are evenly distributed throughout the table. However, because there are only 10 categories, even when selecting charges for only one category, a significant number of rows will be returned. Thus, the index is not beneficial.**

Exercise 2
Viewing Index Statistics and Evaluating Index Selectivity

In this exercise, you will create various indexes on the **Member** table, obtain index statistics, and evaluate whether an index is useful to the query optimizer based on its selectivity.

You can open, review, and execute sections of the IndexStats.sql script file in C:\Moc\SQL2073A\Labfiles\L07, or type and execute the provided Transact-SQL statements.

► To create indexes

In this procedure, you will execute a script that checks for any existing indexes and statistics, drops them, and then creates appropriate indexes. You will view the statistics based on the indexes created.

1. Using SQL Query Analyzer, type and execute this statement to drop existing indexes on the **Member** table:

```
USE credit
EXEC index_cleanup member
```

2. Type and execute these statements to create three indexes on the **Member** table:

```
USE credit
CREATE UNIQUE INDEX indx_member_no ON member (member_no)
CREATE INDEX indx_corp_lname ON member(corp_no,lastname)
CREATE INDEX indx_lastname ON member (lastname)
GO
```

► **To view index statistics and evaluate selectivity of indexes**

In this procedure, you will obtain index statistics for the new indexes, record the statistical information, and evaluate the selectivity of the indexes.

1. Type and execute this statement to display index statistical information on the **member_no** column of the **Member** table:

```
USE credit
DBCC SHOW_STATISTICS (member,indx_member_no)
```

2. Record the statistical information in the following table.

Information	Result
Rows	**10,000**
Steps	**3**
Density	**.00009**
All density	**.00009**

How selective is the index on the **member_no** column?

Very selective. The index_member_no index is created on the member_no column, which contains unique values. If a query specifies a member number in the WHERE clause using an equality, only one row will be returned.

3. Type and execute this statement to display index statistical information on a composite index on the **corp_no** and **lastname** columns of the **Member** table:

```
USE credit
DBCC SHOW_STATISTICS (member,indx_corp_lname)
```

4. Record the statistical information in the following table.

Information	Result
Rows	**10,000**
Steps	**200**
Density	**.0003**
All density (**corp_no**)	**.002**
(**corp_no**, **lastname**)	**.0006**

How selective is this index?

Still selective, but not as selective as member_no column, because duplicate values exist.

5. Type and execute this statement to display index statistical information on the **lastname** column of the **Member** table:

```
USE credit
DBCC SHOW_STATISTICS (member,indx_lastname)
```

6. Record the statistical information in the following table.

Information	Result
Rows	**10,000**
Steps	**26**
Density	**.0**
All density	**.03**

How selective is this index?

This index is not very selective. Using the formula to calculate density shows that this index has low selectivity. Because you cannot divide by zero, zero is the result. All density indicates the total density.

((26/0)/10000) = 0

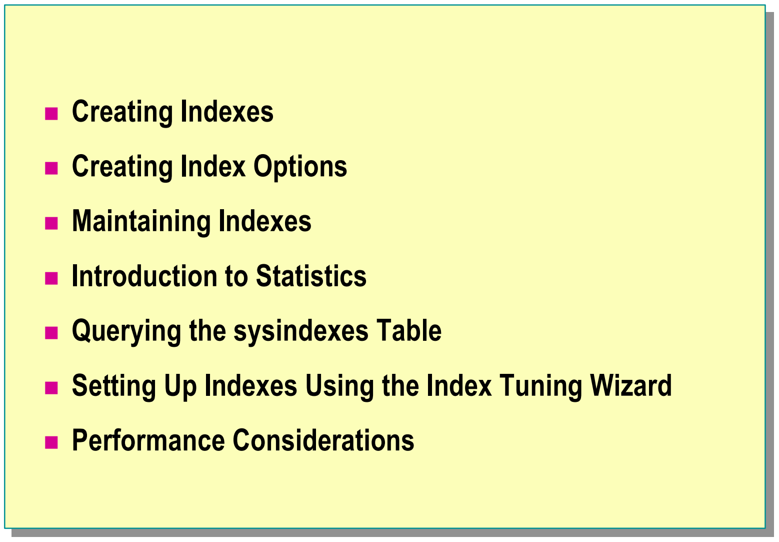
____

____

Review

Topic Objective
To reinforce module objectives by reviewing key points.

Lead-in
The review questions cover some of the key concepts taught in the module.



- Creating Indexes
- Creating Index Options
- Maintaining Indexes
- Introduction to Statistics
- Querying the sysindexes Table
- Setting Up Indexes Using the Index Tuning Wizard
- Performance Considerations

1. You are the database administrator responsible for a large customer database. Lately, the order processing department has been encountering slower system response times when submitting customer orders. Your experience says that the indexing on the **Orders** and **Order Details** tables is correct. What other factors may be causing the slow performance?

   Index statistics may not be automatically maintained; therefore, they may be becoming more and more out of date as data is modified. The FILLFACTOR option may need to be reapplied to allocate table and index space for new orders (rows) that are inserted into the Orders and Order Details tables.

2. What are the advantages of having the SQL Server automatically create and update statistics?

   Letting the query optimizer automatically create and update statistics reduces administrative overhead and improves query performance.

3. You now have the responsibility of maintaining a database that the Sales department uses for taking customer orders. The **Sales** database has performed poorly. Your manager asks you to improve performance in two days. What is the appropriate tool to use to solve this problem?

   Use the Index Tuning Wizard. On the first day, create a workload file recording a full day of user activity. On the second day, run the Index Tuning Wizard against that workload; review the index analysis, and apply the indexes that the Index Tuning Wizard suggests.