MICROSOFT
TRAINING
AND CERTIFICATION

Microsoft Official Curriculum

Module 6: Planning Indexes

Contents

Overview	1
Introduction to Indexes	2
Index Architecture	7
How SQL Server Retrieves Stored Data	12
How SQL Server Maintains Index and Heap Structures	19
Deciding Which Columns to Index	25
Recommended Practices	36
Lab A: Determining the Indexes of a Table	37
Review	41

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:
90 Minutes

Lab:
15 Minutes

This module provides students with an overview of planning indexes. It explains how indexes can improve database performance. It discusses how Microsoft® SQL Server™ 2000 stores clustered and nonclustered indexes and how SQL Server retrieves rows by using indexes. It also explores how SQL Server maintains indexes. The module concludes with guidelines for deciding which columns to index.

In the lab, students explore two methods of determining the indexes on a table.

After completing this module, students will be able to:

- Describe why and when to use an index.
- Describe how SQL Server uses clustered and nonclustered indexes.
- Describe how SQL Server index architecture facilitates the retrieval of data.
- Describe how SQL Server maintains indexes and heaps.
- Describe the importance of selectivity, density, and distribution of data when deciding which columns to index.

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 Microsoft PowerPoint® file 2073A_06.ppt
- The C:\Moc\2073A\Demo\D06_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 lab.

Multimedia Presentation

This section provides multimedia presentation procedures that do not fit in the margin notes or are not appropriate for the student notes.

SQL Server Index Architecture

► **To prepare and start the multimedia presentation**

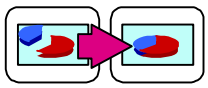
- Click the button in the slide to start the multimedia presentation.

  This multimedia presentation introduces the SQL Server index architecture. It starts with the concept of the *balanced tree* (B-Tree). Then, it discusses the architecture of a clustered index, explains how data is stored, and how data is accessed in a clustered index.

  The presentation continues with the architecture of a nonclustered index, explaining how it differs from a clustered index and pointing out that there are two possible structures of a nonclustered index. Then, it concludes with the demonstration on how data is accessed by using a nonclustered index built on top of a heap and a nonclustered index built on top of a clustered index.

Other Activities

This section provides procedures for implementing interactive activities to present or review information, such as games or role playing exercises.



Displaying the Animated PowerPoint Slides

All animated slides are identified with an icon of links on the lower left corner of the slide.

► **To display the Finding Rows Without Indexes slide**

This slide shows how SQL Server finds rows in a heap when no useful indexes are present.

1. Display the topic slide where the **sysindexes** table, the Index Allocation Map (IAM) page, and data pages are visible. Point out that the data pages are a heap and that there are no nonclustered indexes.
2. Advance to the first animation. Describe how the **FirstIAM** column of the **sysindexes** table points SQL Server to the IAM page.
3. Advance to the final animation that shows how the IAM page directs SQL Server to the extents containing data for that table. Do not discuss the IAM page in detail.

   Point out that table scans are an efficient method of returning all rows in a table.

► To display the Finding Rows in a Heap with a Nonclustered Index slide

This slide shows how SQL Server accesses the range of last names between Masters and Rudd by using a nonclustered index on a heap.

1. Display the topic slide where the architecture of a nonclustered index on a heap and the SELECT statement appear.
2. Advance to the first animation where SQL Server uses the nonclustered index to locate the leaf level of the index with the last name Matey. The index row is selected.
3. Advance to the next animation where SQL Server uses the row identifier to locate the last name Matey in the data pages.
4. Advance to the final animation where the arrow points to the next rows in the leaf level of the index and uses the row identifiers to locate the other rows in the heap.

   Point out that the row identifier uniquely identifies rows that contain identical last names, such as Jones. SQL Server returns these rows in response to the query.

► To display the Finding Rows in a Clustered Index slide

This slide shows how SQL Server finds a row in a clustered index, starting with the **sysindexes** table.

1. Display the topic slide where the architecture of a clustered index and the SELECT statement appear.
2. Advance to the first and only animation where a series of arrows shows the path that SQL Server traverses in the index to find the row for Ota.

   Point out that the leaf level of the clustered index is sorted by the clustering key of **lastname**.

► To display the Finding Rows in a Clustered Index with a Nonclustered Index slide

This slide shows how SQL Server accesses a single row with the first name of Mike by using a nonclustered index on **firstname** and a clustered index on **lastname**.

1. Display the topic slide where the architecture of a nonclustered index on a clustered index and the SELECT statement appear.
2. Advance to the first animation where a series of arrows shows the path in which SQL Server traverses the nonclustered index to find the row with Mike as the first name.
3. Advance to the next animation where a series of arrows shows how SQL Server uses the clustering key, with the last name of Nash, to find the row in the clustered index.

   Point out that SQL Server must traverse two indexes to find the record, but that index retrievals are fast, and upper levels of the index are often in memory.

► To display the Page Splits in an Index slide

This slide shows why and how SQL Server splits a data page.

1. Display the topic slide. Read the Transact-SQL statement and show where the new row goes. Point out that there is no room for the row.
2. Advance the slide showing the page split. Point out that a new page was added and the contents were divided (split) between the old and new pages.

► To display the Forwarding Pointer in a Heap slide

This slide shows why and how SQL Server uses a forward pointer in a heap.

1. Display the topic slide. Read the Transact-SQL statement and point out that the update of the row for Ota will require the row to grow.
2. Advance to the next animation showing how the index is used to locate the row. Point out that the data page is full; there is no room for the row to grow larger.
3. Advance to the final animation that shows how the row is moved to another page. Discuss how the original location keeps a forwarding pointer to the new location. Point out that the nonclustered index still points to the original location.

Module Strategy

Use the following strategy to present this module:

- Introduction to Indexes

  Describe how SQL Server stores and accesses data. Then discuss whether to create indexes. Explain how SQL Server uses indexes.

- Index Architecture

  Play the multimedia presentation on SQL Server index architecture. Avoid discussing the use of indexes in greater detail because the various types of data retrieval are addressed in subsequent sections of this module.

  Discuss how SQL Server uses heaps and present specific points about clustered and nonclustered indexes.

- How SQL Server Retrieves Stored Data

  The illustrations in this section repeat much of the content covered in the multimedia presentation. The amount of time spent discussing each slide should vary based on the knowledge level of the class.

  Introduce the students to **sysindexes**. Point out that the IAM page is used for table or index scans, and that the root is used for drilling down through the indexes. Differentiate between a heap and a table with a clustered index. Make sure that the class understands how a nonclustered index uses either the clustered index or the Row ID (RID) of a heap. Emphasize that a shorter clustered index value can make a more-efficient nonclustered index, but that clustered indexes are selected primarily on the range of queries that are expected.

- How SQL Server Maintains Index and Heap Structures

  Make sure that students understand why pages are split. Emphasize that splits usually do not require index maintenance. Discuss forward pointers and emphasize that they reduce the need to update nonclustered indexes. Review the key points of updates and deletes.

- Deciding Which Columns to Index

  Affirm the importance of creating useful indexes by reviewing the fundamentals of indexes, such as understanding the data and writing queries that limit a search. Then, explain the importance of selecting the appropriate columns to index, because selectivity, density, and distribution of data affect how the query optimizer accesses data.

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 lab in this module is 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 lab in this module.

Setup Requirement

The lab in this module requires the **credit** 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\Restore06.cmd batch file.

Warning If this course has been customized, students must execute the C:\Moc\2073A\Batches\Restore06.cmd batch file to ensure that the first 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 when and why you create indexes and the different types of indexes. You will learn how SQL Server uses and maintains indexes and how to plan the appropriate index for your needs.

- **Introduction to Indexes**
- **Index Architecture**
- **How SQL Server Retrieves Stored Data**
- **How SQL Server Maintains Index and Heap Structures**
- **Deciding Which Columns to Index**

This module provides an overview of planning indexes. It explains how indexes can improve database performance. It discusses how Microsoft® SQL Server™ 2000 stores clustered and nonclustered indexes and how SQL Server retrieves rows by using indexes. It also explores how SQL Server maintains indexes. The module concludes with guidelines for deciding which columns to index.

After completing this module, you will be able to:

- Describe why and when to use an index.
- Describe how SQL Server uses clustered and nonclustered indexes.
- Describe how SQL Server index architecture facilitates the retrieval of data.
- Describe how SQL Server maintains indexes and heaps.
- Describe the importance of selectivity, density, and distribution of data when deciding which columns to index.

Introduction to Indexes

Topic Objective
To introduce indexes.

Lead-in
This section describes why and when to use an index.

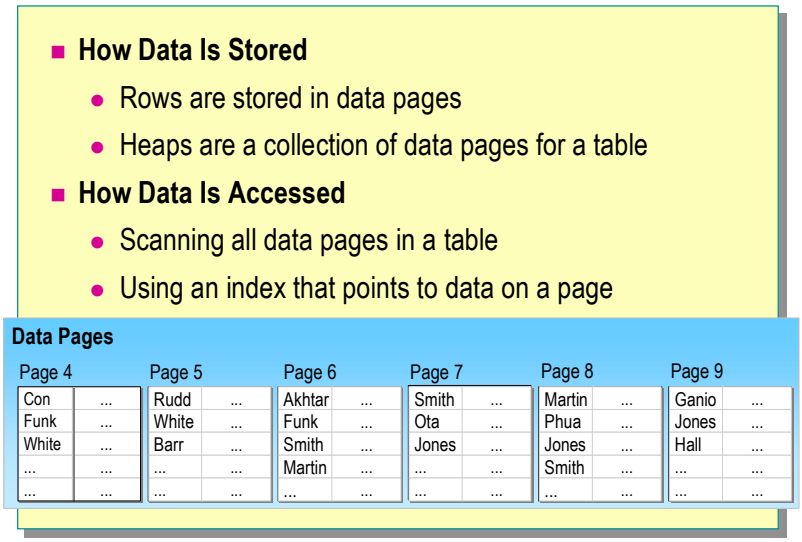
- How SQL Server Stores and Accesses Data
- Whether to Create Indexes

Using indexes can greatly improve database performance. This section introduces basic index concepts and discusses when and why indexes are used.

How SQL Server Stores and Accesses Data

Topic Objective
To discuss how SQL Server stores and accesses data.

Lead-in
Understanding how data is stored is the basis for understanding how SQL Server accesses data.



Understanding how data is stored is the basis for understanding how SQL Server accesses data.

Delivery Tip
Point out that on the slide, only the last names are shown in the data pages, although the data pages store complete rows.

Note In the illustration, only the last names are shown in the data pages, although the data pages store complete rows.

How Data Is Stored

A heap is a collection of data pages containing rows for a table:

- Each data page contains 8 kilobytes (KB) of information. A group of eight adjacent pages is called an *extent*.
- The data rows are not stored in any particular order, and there is no particular order to the sequence of the data pages.
- The data pages are not linked in a linked list.
- When rows are inserted into a page and a page is full, the data pages split.

How Data Is Accessed

SQL Server accesses data in one of two ways:

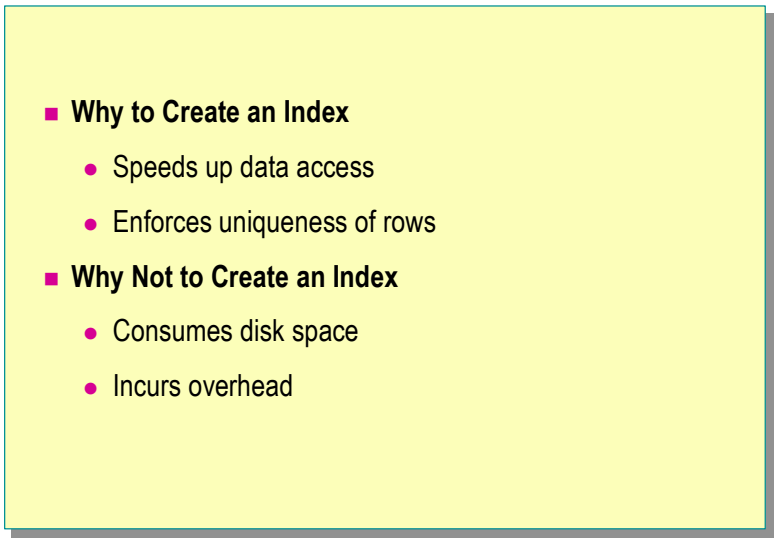
- Scanning all of the data pages of tables—called a *table scan*. When SQL Server performs a table scan, it:
  - Starts at the beginning of the table.
  - Scans from page-to-page through all of the rows in the table.
  - Extracts the rows that meet the criteria of the query.
- Using indexes. When SQL Server uses an index, it:
  - Traverses the index tree structure to find rows that the query requests.
  - Extracts only the needed rows that meet the criteria of the query.

SQL Server first determines whether an index exists. Then, the query optimizer, the component responsible for generating the optimum execution plan for a query, determines whether scanning a table or using the index is more efficient for accessing data.

Whether to Create Indexes

Topic Objective
To discuss whether to create indexes.

Lead-in
Creating an index is not required. Let's discuss why you would want to create an index.



When you are considering whether to create an index, evaluate two factors to ensure that the index will be more efficient than a table scan: the nature of the data and the nature of the queries based on the table.

Why to Create an Index

Indexes accelerate data retrieval. For example, without an index, you would have to go through an entire textbook one page at a time to find information about a topic.

SQL Server uses indexes to point to the location of a row on a data page instead of having to look through all of the data pages of a table. Consider the following facts and guidelines about indexes:

- Indexes generally accelerate queries that join tables and perform sorting or grouping operations.
- Indexes enforce the uniqueness of rows if uniqueness is defined when you create the index.
- Indexes are created and maintained in ascending or descending sorted order.
- Indexes are best created on columns with a high degree of selectivity—that is, columns or combinations of columns in which the majority of the data is unique.

Delivery Tip
Ask: Are indexes required?

Answer: No. You can query and manipulate data without an index. However, data access is considerably slower.

Why Not to Create an Index

Indexes are useful, but they consume disk space and incur overhead and maintenance costs. Consider the following facts and guidelines about indexes:

- When you modify data on an indexed column, SQL Server updates the associated indexes.
- Maintaining indexes requires time and resources. Therefore, do not create an index that you will not use frequently.
- Indexes on columns containing a large amount of duplicate data may have few benefits.

Index Architecture

Topic Objective
To introduce the clustered and nonclustered index architecture.

Lead-in
This section describes how SQL Server uses clustered and nonclustered indexes.

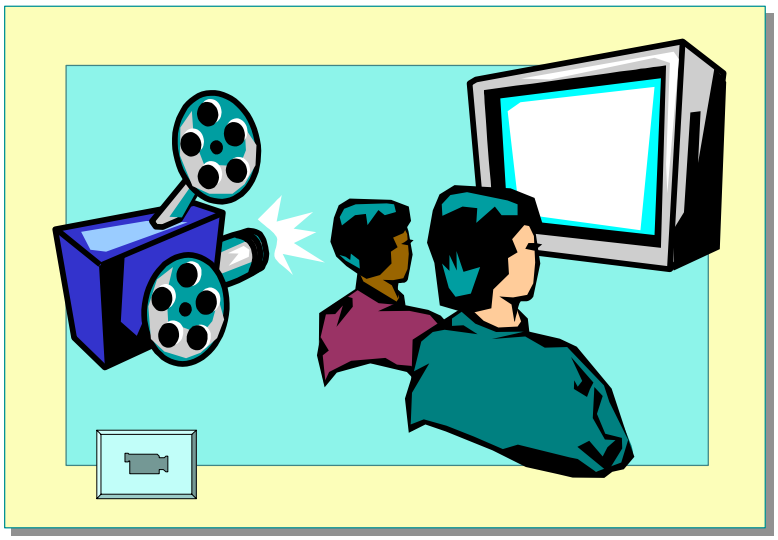
- SQL Server Index Architecture
- Using Heaps
- Using Clustered Indexes
- Using Nonclustered Indexes

The index architecture for clustered and nonclustered indexes is different. Understanding the differences in architecture will help you create the most effective type of index.

Multimedia Presentation: SQL Server Index Architecture

Topic Objective
To introduce SQL Server index architecture.

Lead-in
Let's watch a multimedia presentation on SQL Server index architecture.



The multimedia presentation presents the following concepts.

Clustered Indexes

In a clustered index, the leaf level is the actual data page. Data is physically stored on a data page in ascending order. The order of the values in the index pages is also ascending.

Nonclustered Indexes Built on Top of a Heap

When a nonclustered index is built on top of a heap, SQL Server uses row identifiers in the index pages that point to rows in the data pages. The row identifiers store data location information.

Nonclustered Indexes Built on Top of a Clustered Index

When a nonclustered index is built on top of a table with a clustered index, SQL Server uses a clustering key in the index pages that point to the clustered index. The clustering key stores data location information.

Using Heaps

Topic Objective
To discuss how SQL Server uses heaps.

Lead-in
SQL Server maintains data pages in a heap unless a clustered index is defined on the table.

SQL Server:

- **Uses Index Allocation Map Pages That:**
  - Contain information on where the extents of a heap are stored
  - Navigate through the heap and find available space for new rows being inserted
  - Connect data pages
- **Reclaims Space for New Rows in the Heap When a Row Is Deleted**

SQL Server maintains data pages in a heap unless a clustered index is defined on the table. SQL Server:

- Uses Index Allocation Map (IAM) pages to maintain heaps. IAM pages:
  - Contain information on where the extents of a heap are stored.

    The **sysindexes** system table stores a pointer to the first IAM page associated with a heap.
  - Are used to navigate through the heap and find available space for new rows being inserted.
  - Connect the data pages.

    The data pages and the rows within them are not in any specific order and are not linked together. The only logical connection between data pages is that which is recorded in the IAM pages.
- Reclaims space for new rows in the heap when a row is deleted.

Using Clustered Indexes

Topic Objective
To discuss some facts about clustered indexes.

Lead-in
Clustered indexes are useful for columns that are searched frequently for ranges of key values or are accessed in sorted order.

- **Each Table Can Have Only One Clustered Index**
- **The Physical Row Order of the Table and the Order of Rows in the Index Are the Same**
- **Key Value Uniqueness Is Maintained Explicitly or Implicitly**

Delivery Tip
After discussing the bullet points on the slide, ask students what happens to a clustered index when rows are added to a table.

Ask: Why can't you have two clustered indexes on a table?

Answer: SQL Server only stores one physical ordering of rows for a table.

Clustered indexes are useful for columns that are searched frequently for ranges of key values, or are accessed in sorted order. When you create a clustered index, consider the following facts and guidelines:

- Each table can have only one clustered index.
- The physical row order of the table and the order of rows in the index are the same. You should create clustered indexes before you create any nonclustered indexes because a clustered index changes the physical row order of the table. Rows are sorted into a sequenced order and maintained in that order.
- Key value uniqueness is maintained explicitly, with the UNIQUE keyword, or implicitly, with an internal unique identifier. These unique identifiers are internal to SQL Server and are not accessible to the user.
- The average size of a clustered index is about five percent of the table size. However, clustered index size varies depending on the size of the indexed column.
- When a row is deleted, space is reclaimed and is available for a new row.
- During index creation, SQL Server temporarily uses disk space from the current database. A clustered index requires about 1.2 times the table size for working space when the index is created. The disk space that is used during index creation is reclaimed automatically after the index is created.

Delivery Tip
Ask: Where does the 1.2 value come from?

Answer: 1 = data and .2 = index. These values are conservative estimates.

Note Be sure that you have sufficient disk space in the database when you create clustered indexes.

Using Nonclustered Indexes

Topic Objective
To discuss some facts about nonclustered indexes.

Lead-in
Nonclustered indexes are useful when users require multiple ways to search data.

- **Nonclustered Indexes Are the SQL Server Default**
- **Existing Nonclustered Indexes Are Automatically Rebuilt When:**
  - An existing clustered index is dropped
  - A clustered index is created
  - The DROP_EXISTING option is used to change which columns define the clustered index

Nonclustered indexes are useful when users require multiple ways to search data. For example, a reader may frequently search through a gardening book, looking for both the common and scientific names of plants. You could create a nonclustered index for retrieving the scientific names and a clustered index for retrieving common names. When you create a nonclustered index, consider the following facts and guidelines:

- If an index type is not specified, the type will default to nonclustered index.
- SQL Server automatically rebuilds existing nonclustered indexes when any of the following occurs.
  - An existing clustered index is dropped.
  - A clustered index is created.
  - The DROP_EXISTING option is used to change which columns define the clustered index.
- The order of the leaf level pages of a nonclustered index differs from the physical order of the table. The leaf level is sorted in ascending order.
- Uniqueness is maintained at the leaf level with either clustering keys or row identifiers.
- You can have up to 249 nonclustered indexes per table.
- Nonclustered indexes are best created on columns in which the data selectivity ranges from highly selective to unique.
- Create clustered indexes before nonclustered indexes.
- The row identifiers specify the logical ordering of rows and consist of the file ID, page number, and row ID.

How SQL Server Retrieves Stored Data

Topic Objective
To introduce data retrieval.

Lead-in
This section describes how SQL Server index architecture facilitates the retrieval of data.

- How SQL Server Uses the sysindexes Table
- Finding Rows Without Indexes
- Finding Rows in a Heap with a Nonclustered Index
- Finding Rows in a Clustered Index
- Finding Rows in a Clustered Index with a Nonclustered Index

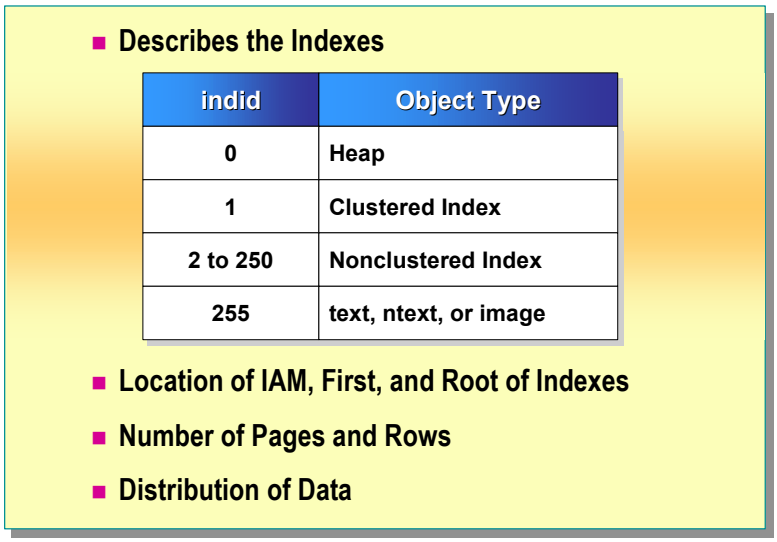
This section repeats information contained in the prerequisite courses. Review it rapidly if students have a good understanding of a SELECT statement.

To design efficient databases, it is important to understand how SQL Server retrieves stored data. This section describes how SQL Server index architecture facilitates the retrieval of data.

How SQL Server Uses the sysindexes Table

Topic Objective
To describe the role of **sysindexes** in a data search.

Lead-in
The **sysindexes** system table provides the first step in a data search.



The **sysindexes** system table is a central location for vital information about tables and indexes. It contains statistical information, such as the number of rows and data pages in each table. It describes how to find information stored in a data table.

Page pointers in the **sysindexes** table anchor all of the page collections for tables and indexes. Every table has one collection of data pages, plus additional collections of pages to implement each index defined for the table.

A row in **sysindexes** for each table and index is uniquely identified by the combination of the object identifier column (**id**) and the index identifier column (**indid**).

The indid Column

This is how the columns of the **sysindexes** table assist in locating data pages for different types of objects:

- A heap has a row in **sysindexes** with the **indid** column set to zero. The **FirstIAM** column in **sysindexes** points to the chain of IAM pages for the collection of data pages in the table. SQL Server must use the IAM pages to find the pages in the data page collection because these pages are not linked together.
- A clustered index created for a table has a row in **sysindexes** with the **indid** column set to 1. The **root** column in **sysindexes** points to the top of the clustered index *balanced tree* (B-Tree).

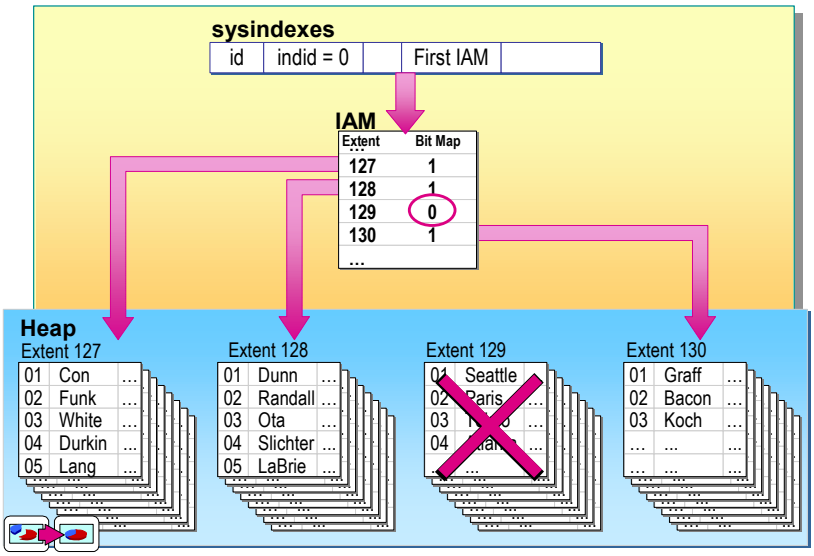
- Each nonclustered index created for a table has a row in **sysindexes** with a value in the **indid** column**.** The value for the **indid** column for a nonclustered index ranges from 2 to 250. The **root** column in **sysindexes** points to the top of the nonclustered index B-Tree.
- Each table that has at least one **text, ntext**, or **image** column also has a row in **sysindexes** with the **indid** column set to 255. The **FirstIAM** column in **sysindexes** points to the chain of IAM pages that manage the **text**, **ntext**, and **image** pages.

Delivery Tip
Information generated by the UPDATE STATISTICS command is also held in sysindexes.

Finding Rows Without Indexes

Topic Objective
To describe data searching in a heap.

Lead-in
When a table has neither a clustered index nor useful nonclustered indexes, SQL Server uses the IAM page to initiate a table scan.



Delivery Tip
This slide is animated. Refer to the Instructor Notes if you require help with navigating through this slide.

Point out that the IAM page is often in memory and contains efficient, densely packed information.

Point out that only a table scan can retrieve rows in the absence of an index.

When an index does not exist on a table, SQL Server must use a table scan to retrieve rows. SQL Server uses the **sysindexes** table to find the IAM page. Because the IAM page contains a list of all pages related to that table, as a bitmap of eight-page extents, SQL Server can then read all data pages.

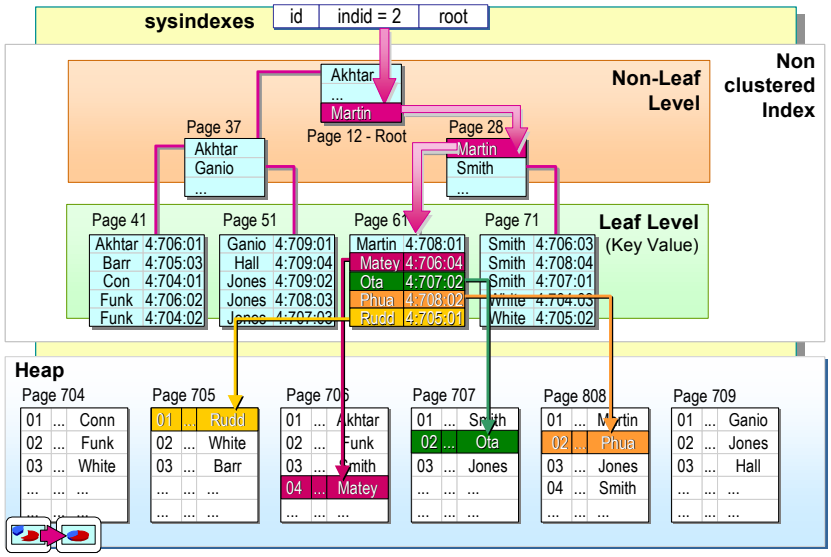
Initiating a data search on a heap by using an IAM page is efficient for a table scan, but it is not a good means to find a small number of rows in a large table.

The rows are returned unsorted. They may initially be returned in insertion order, but this order will not be maintained. After deletions have occurred, new inserts will fill in the gaps, making the order unpredictable.

Finding Rows in a Heap with a Nonclustered Index

Topic Objective
To describe data searching with a nonclustered index.

Lead-in
Pointers are critical to nonclustered index searching.



Delivery Tip
This slide is animated. Refer to the Instructor Notes if you require help in navigating through this slide.

Students should already be familiar with the B-Tree structure. Emphasize the use and structure of the pointers.

A nonclustered index is like the index of a textbook. The data is stored in one place and the index is stored in another. Pointers indicate the storage location of the indexed items in the underlying table.

SQL Server indexes are organized as B-Trees. Each page in an index holds a page header followed by index rows. Each index row contains a key value and a pointer to either another page or a data row.

Each page in an index is called an index node. The top node of the B-Tree is called the root node or root level. The bottom node is called the leaf node or leaf level. Any index levels between the root and the leaf nodes are intermediate levels. Each page in the intermediate or bottom layers has a pointer to the preceding and subsequent pages in a doubly linked list.

In a table that only contains a nonclustered index, the leaf nodes contain row locators with pointers to the data rows holding the key values. Each pointer (Row ID or RID) is built from the file ID, page number, and the number of the row on the page.

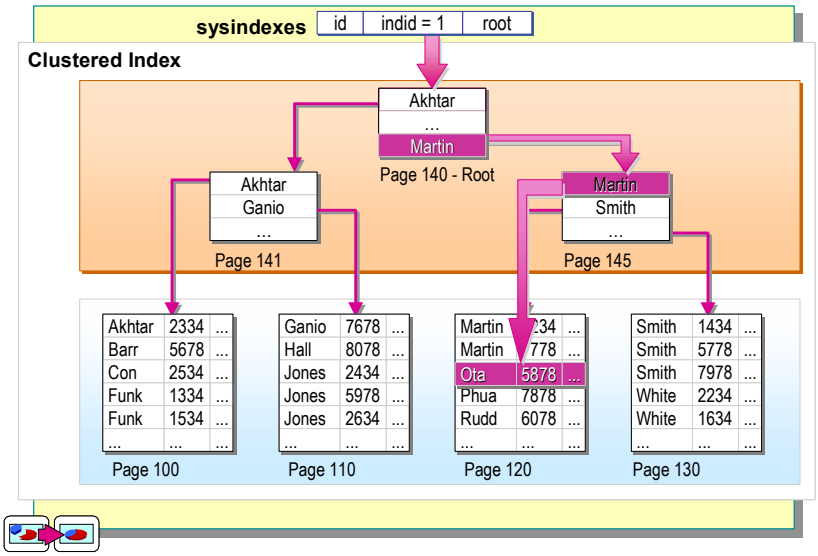
Example

```
SELECT lastname, firstname
FROM member
WHERE lastname
BETWEEN 'Masters' AND 'Rudd'
```

Finding Rows in a Clustered Index

Topic Objective
To describe data searching with a clustered index.

Lead-in
Clustered indexes are sorted in sequential order based on their clustered key.



Delivery Tip
This slide is animated. Refer to the Instructor Notes if you require help in navigating through this slide.

Clustered and nonclustered indexes share a similar B-Tree structure. The differences are that:

- The data pages of a clustered index are the leaf nodes of the B-Tree structure.
- The data rows of a clustered index are sorted and stored in a sequential order based on their clustered key.

A clustered index is like a telephone directory in which all of the rows for customers with the same last name are clustered together in the same part of the book. Just as the organization of a telephone directory makes it easy for a person to search, SQL Server quickly searches a table with a clustered index. Because a clustered index determines the sequence in which rows are stored in a table, there can only be one clustered index for a table at a time.

Keeping your clustered key value small increases the number of index rows that can be placed on an index page and decreases the number of levels that must be traversed. This minimizes I/O.

Note If there are duplicate values in a clustered index, SQL Server must distinguish between rows that contain identical values in the key column or columns. It does this by using a 4-byte integer (*uniquifier* value) in an additional system-only uniquifier column.

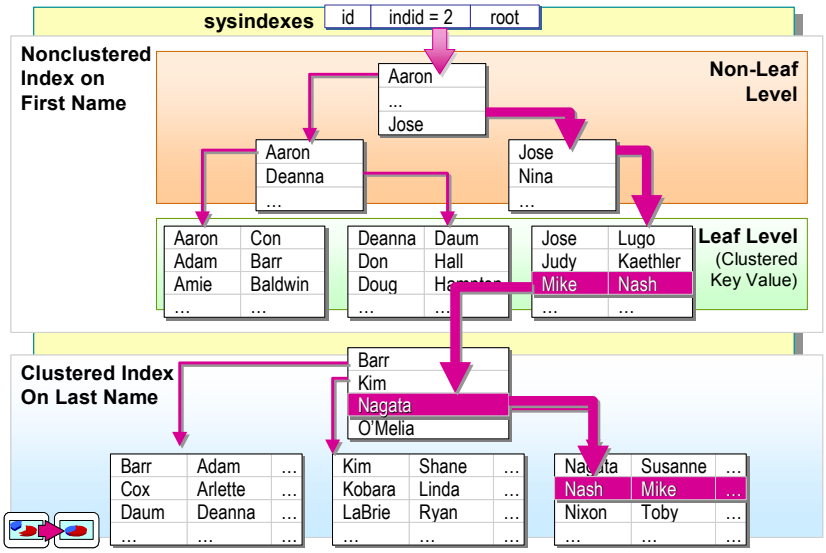
Example

```
SELECT lastname, firstname
FROM member
WHERE lastname = 'Ota'
```

Finding Rows in a Clustered Index with a Nonclustered Index

Topic Objective
To describe data searching when both types of indexes are present.

Lead-in
The secondary index is used to speed up searches on additional columns.



Delivery Tip
This slide is animated. Refer to the Instructor Notes if you require help in navigating through this slide.

When a nonclustered index is added to a table that already has a clustered index, the row locator of each nonclustered index contains the clustered key index value for the row.

When using clustered and nonclustered indexes on the same table, the B-Tree structures of both indexes must be traversed to reach data. This generates additional I/O.

Because the key value of a clustered index is usually larger than the 8-byte RID used for heaps, nonclustered indexes can be substantially larger on clustered indexed tables than when built on heaps. Keeping the key values of the clustered index small helps you to build smaller, faster indexes.

Example

```
SELECT lastname, firstname, phone
FROM member
WHERE firstname = 'Mike'
```

How SQL Server Maintains Index and Heap Structures

Topic Objective
To introduce how SQL Server maintains index and heap structures.

Lead-in
This section describes how SQL Server maintains indexes and heaps.

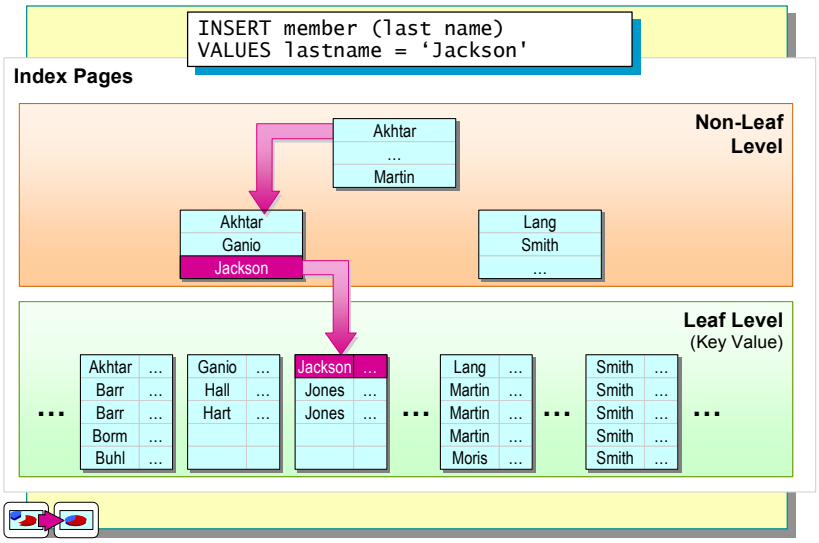
- Page Splits in an Index
- Forwarding Pointer in a Heap
- How SQL Server Updates Rows
- How SQL Server Deletes Rows

This section discusses how SQL Server maintains indexes and heaps when inserting, updating, and deleting rows.

Page Splits in an Index

Topic Objective
To describe the concept of a page split.

Lead-in
Inserting a row into a full page can cause a page split.



Delivery Tip
This slide is animated. Refer to the Instructor Notes if you require help with navigating through this slide.

A clustered index directs an inserted or updated row to a specific page, which is determined by the clustered key value. If either the data page or index page does not have enough room to accommodate the data, a new page is added in a process known as a *page split*. Approximately half of the data remains on the old page, and the other half is moved to the new page.

Logically, the new page follows the original page; physically, the new page may be assigned to any available page. If an index experiences a large number of page splits, rebuilding the index will improve performance.

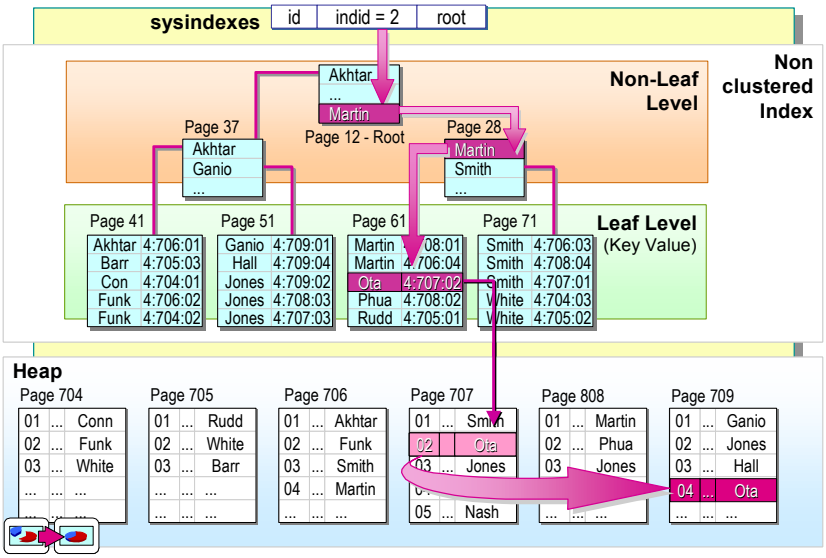
Delivery Tip
Point out that the nonclustered index must be modified to add Jackson, but it does not need to be updated with the new location of Jones.

Note If a page splits in a clustered index, SQL Server does not need to maintain the nonclustered indexes for all of the rows that have moved to a new page. The row locator continues to identify the correct location in the clustering key.

Forwarding Pointer in a Heap

Topic Objective
To describe the concept of a forward pointer.

Lead-in
If a row in a heap grows too large for its original location, it will be moved to another page.



Delivery Tip
This slide is animated. Refer to the Instructor Notes if you require help with navigating through this slide.

Page splits do not occur in a heap. SQL Server has a different means of handling updates and inserts when data pages are full.

Inserts to a Heap

The insert of a new row into a heap cannot cause a page split, because a new row can be inserted wherever room is available.

Forwarding Pointers

If an update to a row in a heap needs more room than is currently available on that page, the row will be moved to a new data page. The row leaves a *forwarding pointer* in its original location. If the row with the forwarding pointer must move again, the original pointer is redirected to the new location.

The forwarding pointer ensures that nonclustered indexes need not be changed. If an update causes the forwarded row to shrink enough to fit in its original place, the pointer is eliminated, and the record is restored to its original location by the update.

Page Splits in Nonclustered Indexes on a Heap

Although an insert or update cannot cause a page split in a heap, if a nonclustered index exists on the heap, an insert or update can still cause a page split in the nonclustered index.

How SQL Server Updates Rows

Topic Objective
To describe the effects of updating data.

Lead-in
Updates can often take place without impacting the structure of data rows.

- **An Update Generally Does Not Cause a Row to Move**
- **An Update Can Be a Delete Followed by an Insert**
- **Batch Updates Touch Each Index Only Once**

Updates can often take place without impacting the structure of data rows.

An Update Generally Does Not Cause a Row to Move

Updates generally do not require rows to move. No move occurs if the update does not enlarge the record or if any enlargement still fits on the same page. Updates typically generate only a single log record.

An Update Can Be a Delete Followed by an Insert

An update causing a row to be moved is logged as a delete followed by an insert, if:

- The update does not fit on a page of a heap.
- The table has an update trigger.
- The table is marked for replication.
- The value of the clustered index key requires the row to be placed in a different location. For example, a last name changed from Abercrombie to Yukish would move that name in a telephone directory.

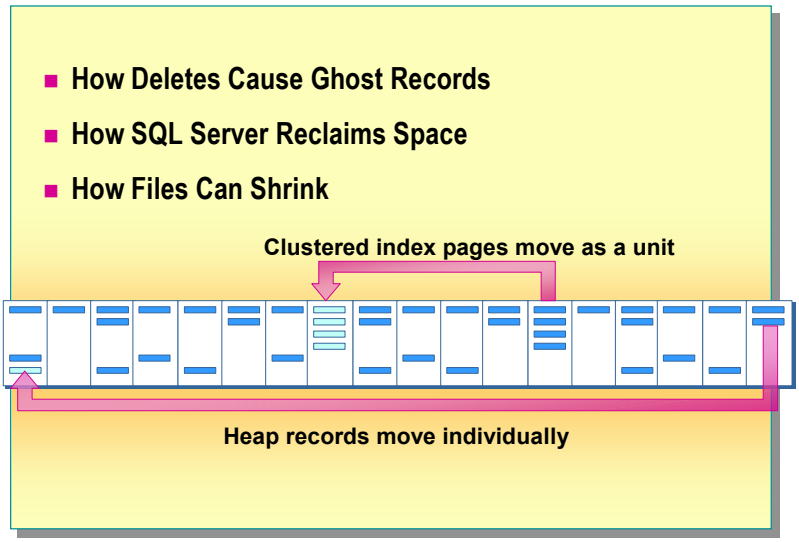
Batch Updates Touch Each Index Only Once

If a significant number of rows are inserted, updated or deleted in a table in a single SQL statement, SQL Server presorts the changes for each index so that the changes are performed in the order of the index. This batch update results in a significantly greater performance improvement than when using a series of Transact-SQL statements.

How SQL Server Deletes Rows

Topic Objective
To describe the deletion of rows.

Lead-in
When a row is deleted, both the index and data pages are changed.



The deletion of rows impacts both the index and data pages.

How Deletes Cause Ghost Records

Rows deleted from the leaf level of an index are not removed immediately. They are marked as invalid and called *ghost records*. This process can prevent the need to lock adjacent records. It can also prevent lock contention over ranges of data. SQL Server periodically initializes a special housekeeping thread that checks indexes for the existence of ghost records and removes them.

How SQL Server Reclaims Space

When the last row is deleted from a data page, the entire page is deallocated, unless it is the only page remaining in the table.

Deleting Rows in an Index

Space in an index is available for use by adjacent rows immediately after a row is deleted, but some gaps usually remain until the index is rebuilt.

Deleting Rows in a Heap

Deleted rows in heaps are not compacted until the space is required for an insertion.

How Files Can Shrink

Delivery Tip
Point out that rows in a heap move individually. Rows do not stay in insertion order.

After records have been deleted, a file is able to shrink. SQL Server shrinks a file by moving data to available pages at the beginning of the file. Within an index, SQL Server moves whole pages so that the rows stay in their proper, sorted relationship. Page pointers adjust to link the moved page into the correct sequence in the table. If there is no clustered index, individual rows can move wherever there is room in the file.

Note A database option, **autoshrink**, tries to shrink the database without manual intervention. It does so five minutes after startup and every thirty minutes thereafter. The file is shrunk to a size where 25 percent of the file is unused space, or to the size of the file when it was created, whichever is greater.

Deciding Which Columns to Index

Topic Objective
To point out the topics in this section.

Lead-in
Planning useful indexes is one of the most important aspects of improving query performance.

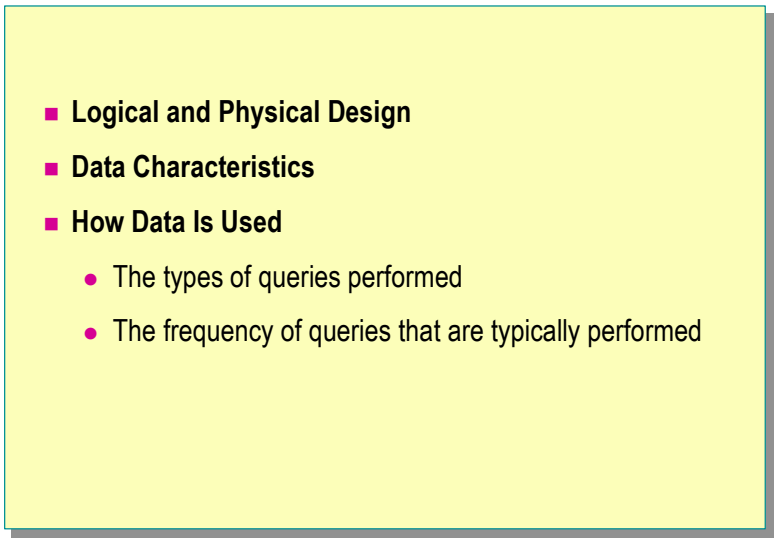
- Understanding the Data
- Indexing Guidelines
- Choosing the Appropriate Clustered Index
- Indexing to Support Queries
- Determining Selectivity
- Determining Density
- Determining Distribution of Data

Planning useful indexes is one of the most important aspects of improving query performance. It requires both an understanding of index structure and how the data is used.

Understanding the Data

Topic Objective
To point out that the first step in creating indexes is to understand the data and how users access it.

Lead-in
Before you create an index, you should have a thorough understanding of the data.



Before you create an index, you should have a thorough understanding of the data, including:

- Logical and physical design.
- Data characteristics.
- How data is used.

  To design useful and effective indexes, you must rely on the analysis of queries that users send. A poor analysis of how users access data becomes apparent in the form of slow query response or even unnecessary table locks. You should be aware of how users access data by observing:

  - The types of queries performed.
  - The frequency of queries that are typically performed.

Having a thorough understanding of the user's data requirements helps to determine which columns to index and what types of indexes to create. You might have to sacrifice some speed on one query to gain better performance on another.

Indexing Guidelines

Topic Objective
To consider which columns to index.

Lead-in
When you create an index, think about the nature of your environment and how data will be distributed.

- **Columns to Index**
  - Primary and foreign keys
  - Those frequently searched in ranges
  - Those frequently accessed in sorted order
  - Those frequently grouped together during aggregation
- **Columns Not to Index**
  - Those seldom referenced in queries
  - Those that contain few unique values
  - Those defined with **text**, **ntext**, or **image** data types

Your business environment, data characteristics, and use of the data determine the columns that you specify to build an index. The usefulness of an index is directly related to the percentage of rows returned from a query. Low percentages or high selectivity are more efficient.

Note When you create an index on a column, the column is referred to as the index column. A value within an index column is called a key value.

Delivery Tip
Ask: Can every column be indexed?

Answer: Yes, every column can be indexed, but doing so would be inefficient.

Ask: Can you index a single column more than once?

Answer: Yes, but doing so is generally inefficient.

Columns to Index

Create indexes on frequently searched columns, such as:

- Primary keys.
- Foreign keys or columns that are used frequently in joining tables.
- Columns that are searched for ranges of key values.
- Columns that are accessed in sorted order.
- Columns that are grouped together during aggregation.

Columns Not to Index

Do not index columns that:

- You seldom reference in a query.
- Contain few unique values. For example, an index on a column with two values, male and female, returns a high percentage of rows.
- Are defined with **text**, **ntext**, and **image** data types. Columns with these data types cannot be indexed.

Choosing the Appropriate Clustered Index

Topic Objective
To determine the best type of index for a database.

Lead-in
Consider how the table is used when you select the clustered index.

- **Heavily Updated Tables**
  - A clustered index with an identity column keeps updated pages in memory
- **Sorting**
  - A clustered index keeps the data pre-sorted
- **Column Length and Data Type**
  - Limit the number of columns
  - Reduce the number of characters
  - Use the smallest data type possible

Delivery Tip
Tell students that they should not automatically place the clustered index on the primary key. Consider the usage of the table.

Consider how the table is used when you choose the clustered index for each table.

Heavily Updated Tables

When you optimize performance for data insertion to a heavily used table, consider creating a clustered index on a primary key identity column. By forcing inserts to a small group of pages at the end of the table, speed increases. The frequent access keeps these pages in memory.

Sorting

Tables that are frequently sorted for reports, grouped for aggregation, or searched for ranges of data can benefit from a clustered index on the sorting column. Using a clustered index is particularly helpful when many columns of the table are returned and a nonclustered index is impractical. For example, a mailing list table would benefit from a clustered index on the postal code, because the mailing labels must be printed and applied in a specified order.

Column Length and Data Type

SQL Server uses the clustered index value as the row identifier within each nonclustered index. The clustered index value can be repeated many times in your table structure.

To prevent large clustered indexes from making their associated nonclustered indexes larger and slower:

- Limit the number of columns in your clustered index.
- Reduce the average number of characters by using a **varchar** data type instead of a **char** data type.
- Use the smallest data type possible, such as **tinyint** instead of **int**.

Indexing to Support Queries

Topic Objective
To understand why it is important to limit the search and how to limit a search.

Lead-in
Query performance relies on how well you design your indexes and how selective your queries are. You should always write queries that limit a search.

- **Using Search Arguments**
- **Writing Good Search Arguments**
  - Specify a WHERE clause in the query
  - Verify that the WHERE clause limits the number of rows
  - Verify that an expression exists for every table referenced in the query
  - Avoid using leading wildcards

Query performance is dependent on how well you have designed your indexes. It is also important to write your queries with a search argument that can take advantage of an indexed column.

Using Search Arguments

A search argument limits a search to an exact match, a range of values, or a combination of two or more items joined by an AND operator. A search argument contains a constant expression that acts on a column by using an operator. When you write queries that contain search arguments, you increase the opportunity for the query optimizer to use an index.

Writing Good Search Arguments

If an expression does not limit a search, it is considered a non-search argument. In many cases, you should rewrite queries to convert non-search arguments into search arguments.

To limit the search, you should:

- Specify a WHERE clause in the query.
- Verify that the WHERE clause limits the number of rows.
- Verify that an expression exists for every table referenced in the query.
- Avoid using leading wildcard characters.

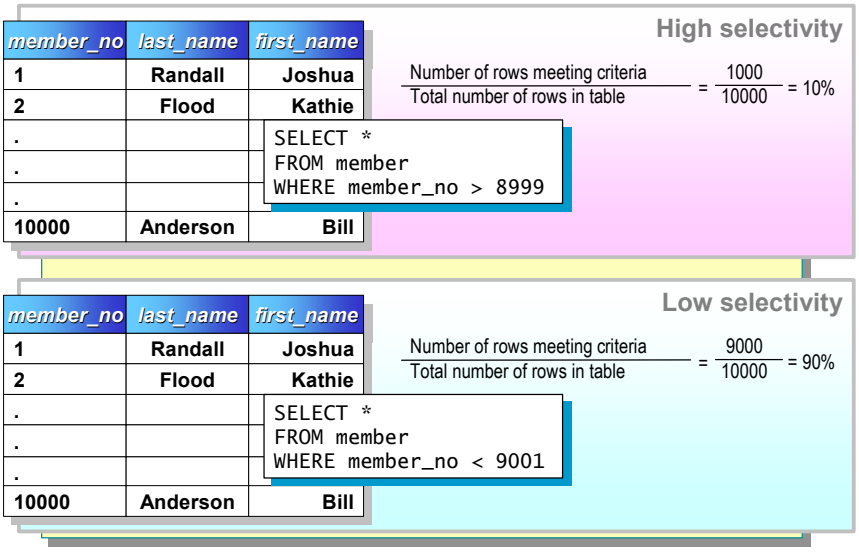
The following table shows good search arguments:

Good search argument	Query
`WHERE cust_id = 47635`	Limits the search because **cust_id** is unique
`WHERE date BETWEEN '07/23/2000' AND '07/30/2000'`	Limits the search to only a small range of data
`WHERE lastname LIKE 'Gre%'`	Limits the search to only last names that begin with the letters Gre

Determining Selectivity

Topic Objective
To discuss the concept of selectivity.

Lead-in
A concept and term that is frequently used when discussing indexes is selectivity.



Delivery Tip
The examples on the slide cannot be executed against the **credit** database.

A concept and term that is frequently used when discussing indexes is *selectivity*. When determining which columns to index and choosing the type of index to create, you should consider how selective the data values are.

Defining Selectivity

Selectivity is derived from the percentage of rows in a table that are accessed or returned by a query. The query optimizer determines selectivity for SELECT, UPDATE, or DELETE statements. When creating indexes, you should create them on:

- Columns that are often referenced in join operations or in the WHERE clause.
- Data that is highly selective.

High Selectivity and Low Selectivity

In *high selectivity*, the search criteria limit the number of rows returned to a low percentage of the total possible. One row returned is the highest selectivity that can be achieved.

In *low selectivity*, the search criteria return a high percentage of the rows in the table.

Estimating Selectivity

You can determine how selective a query is by estimating the number of rows returned, relative to the total number of rows in a table for a specific query.

Example 1

In this example, assuming that there are 10,000 rows in the **member** table and that the member numbers are in the range of 1 to 10,000—all unique values—the query returns one row.

```
SELECT *
FROM member
WHERE member_no = 8999
```

Example 2

In this example, assuming that there are 10,000 rows in the **member** table and that the member numbers are in the range of 1 to 10,000—all unique values—the query returns 999 rows.

```
SELECT *
FROM member
WHERE member_no > 9001
```

Example 3

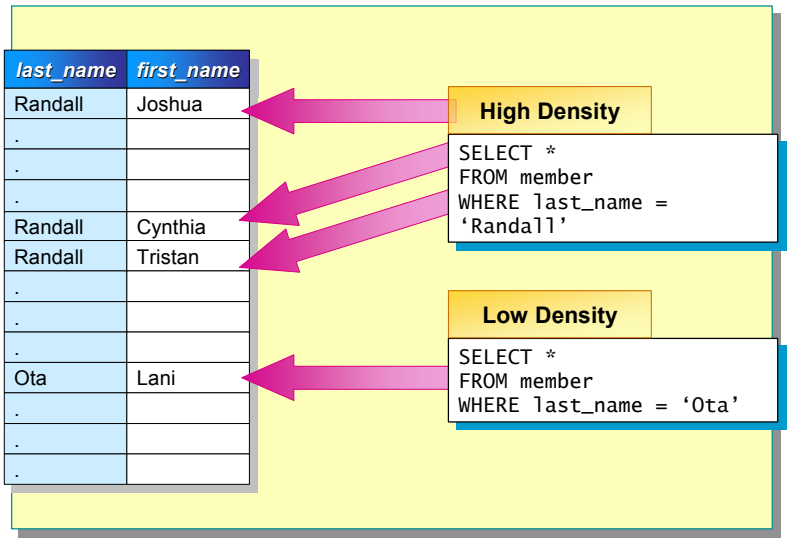
In this example, assuming that there are 10,000 rows in the **member** table and that the member numbers are in the range of 1 to 10,000—all unique values—the query returns 9,000 rows.

```
SELECT *
FROM member
WHERE member_no < 9001
```

Determining Density

Topic Objective
To discuss the concept of density.

Lead-in
A related concept to selectivity is density. When determining which columns to index, you should examine the density of your data.



Delivery Tip
The examples on the slide cannot be executed against the **credit** database.

A related concept to selectivity is the concept of *density*. When determining which columns to index, you should examine the density of your data.

Defining Density

Density is the average percentage of duplicate rows in an index. If your data or query is not very selective (low selectivity), you have a high amount of density.

- An index with a large number of duplicates has high density.

  For example, an index on the **lastname** column can be very dense.

- A unique index has low density.

  An example of this would be an index on social security number, ID, driver's license number, or last name and first name (composite).

Relating Density to the Data

When determining the density of your data, remember that density relates to the specific data elements. Density can vary. Consider an index on the **lastname** column. Data elements of this index are very dense for popular last names such as Randall, whereas the last name Ota is not likely to be very dense.

How Density Affects the Query Plan

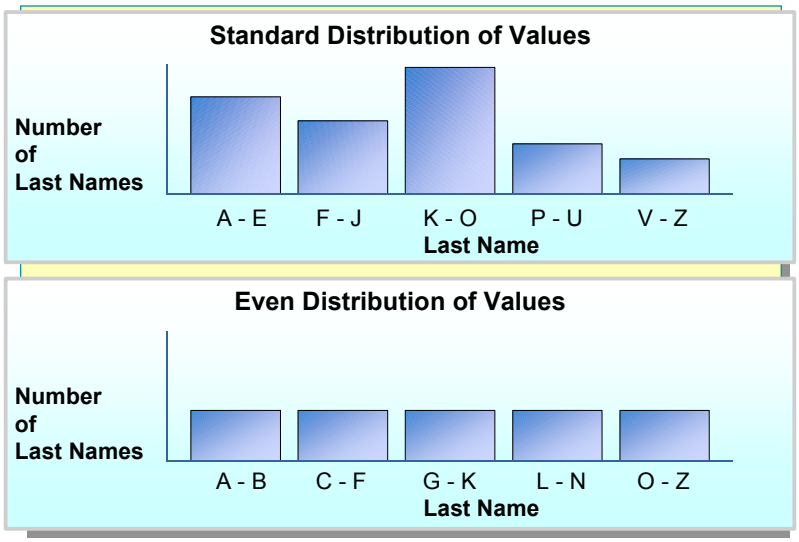
Because data is not distributed evenly, the query optimizer might or might not use an index. In the illustration, the query optimizer might:

- Perform a table scan to retrieve the last name Randall.
- Use an index to access the last name Ota.

Determining Distribution of Data

Topic Objective
To discuss the concept of data distribution and the different ways in which data distribution can be examined.

Lead-in
The distribution of data is related to the concept of density. When determining the density of the data, you should also examine how the data is distributed.



The distribution of data is related to the concept of density. When determining the density of the data, you should also examine how the data is distributed.

Defining Distribution of Data

The *distribution of data* indicates the amount of data that exists for a range of values in a given table and how many rows fall in that range. If an indexed column has very few unique values, data retrieval may be slow because of the distribution of data. For example, a telephone directory sorted alphabetically on last name may show that there is a high occurrence of people with the last name Randall or Jones.

Standard or Even Distribution

In a standard distribution, the key value ranges remain constant while the number per range changes. An even distribution allows the query optimizer to easily determine the selectivity of a query by estimating the number of qualifying rows as a percentage of the total rows in the table.

Relating Density to Distribution of Data

Similar to density, data elements of the index can vary in how the data is distributed. Typically, data is not evenly distributed. For example, if the **member** table contains 10,000 rows and has an index on the **lastname** column, the last names are typically not evenly distributed.

Estimating the Percentage of Rows Returned

In many cases, you can approximate the percentage of data to be returned in a result set. For example, if the criterion is male/female, the result set for females can be estimated at 50 percent. When estimating the percentage of rows returned on values such as last name, city, or other demographic data, it is critical that you know your data, because data distribution varies widely in different environments.

Example

This query is used to show the distribution (amount of duplicates) of column values on an existing database. In this example, the query returns each value only once with a number (count) that indicates how many times it occurs in the table.

Delivery Tip
The examples on the slide cannot be executed against the **credit** database.

```
SELECT column, count(*) AS 'Data Count'
FROM table
GROUP BY column
ORDER BY 'Data count' DESC
```

Recommended Practices

Topic Objective
To present recommended practices for planning indexes.

Lead-in
These are recommended practices for planning indexes.

The following recommended practices should help you plan indexes effectively:

- Create indexes on columns that join tables, including primary keys or foreign keys.
- Create unique indexes to enforce uniqueness in a column or group of columns.
- Review your indexes and drop those that are not being used. Indexes require disk space and time to maintain them. Databases with high data insert activity should have fewer indexes. Databases with high read activity should have more indexes.
- Avoid including unnecessary columns in the clustered index. When possible, use small data types, such as **varchar** instead of **char**.
- Consider using a clustered index to support sorting and range searches. When you optimize a table for data retrieval, the clustered index should support the retrieval of groups of records. Select the column or columns for the clustering key that sorts data in a frequently needed order or that groups records that must be accessed together.
- Create indexes that support the search arguments of common queries. Highly selective columns are good candidates for indexes. Columns with high density are poor candidates for indexes.

Lab A: Determining the Indexes of a Table

Topic Objective
To introduce the lab.

Lead-in
In this lab, you will use two methods to determine the index structure of a table.

Explain the lab objectives.

Objectives

After completing this lab, you will be able to:

- Use the **sp_help** system stored procedure to determine the index structure of a table.
- Query the **sysindexes** table to identify the index structure of a table.

Prerequisites

Before working on this lab, you must have script files for this lab, which are located in C:\Moc\2073A\Labfiles\L06.

Lab Setup

To complete this lab, you must have either:

- Completed the prior lab, or
- Executed the C:\Moc\2073A\Batches\Restore06.cmd batch file.

  This command file restores the **credit** 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: 15 minutes

Exercise 1
Identifying Indexes Using sp_help

In this exercise, you will use the **sp_help** system stored procedure to determine the index structure of a table.

► To use sp_help

In this procedure, you will use the **sp_help** system stored procedure to determine the name, type, and key columns of the indexes on a table.

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 Microsoft 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 **credit**.

4. Open C:\Moc\2073A\Labfiles\L06\Inspect_corporation.sql, and then review and execute it.

   This script will use **sp_help** to return information about the **corporation** table. This stored procedure returns nine grids of data.

5. Navigate to the sixth grid, called **index_name**.

   What are the names of the indexes on the **corporation** table?

   The corporation table has two indexes called corporation_ident and corporation_region_link.

   Is the primary key of the **corporation** table a clustered or a nonclustered index?

   The primary key is a clustered index called corporation_ident.

Exercise 2
Viewing Entries in the sysindexes Table

In this exercise, you will query the **sysindexes** system table and identify indexes.

► **To view the sysindexes system table**

In this procedure, you will execute a script that queries the **sysindexes** system table.

1. Open C:\Moc\2073A\Labfiles\L06\Inspect_sysindexes.sql, and then review and execute it.

   This script will query the **sysobjects** and **sysindexes** tables for user-created tables, sorted by the table name.

   Which tables do not have a clustered index? How can you tell?

   The charge, member, provider and status tables do not have clustered indexes. The indid value of 0 indicates a heap.

   What type of indexes does the **corporation** table have?

   Both clustered and nonclustered indexes.

2. Review the column names in the **sysindexes** table.

   How many rows are in the **member** table? How many pages are used?

   The value of the rows column is 10000. The value of the used column is 192 pages.

   How does SQL Server locate the root of an index or the first IAM page in a heap?

   This information is in the root and FirstIAM columns of sysindexes.

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.

- Introduction to Indexes
- Index Architecture
- How SQL Server Retrieves Stored Data
- How SQL Server Maintains Index and Heap Structures
- Deciding Which Columns to Index

1. If a customer table has no indexes, how does SQL Server find the row for a customer named Eva Corets?

   SQL Server must perform a table scan, reading every row in the table, to find the required row.

2. How many clustered indexes can be created for a table?

   One. The clustered index defines the physical storage of the data pages, and the complete table data is stored in only one location.

3. How does a nonclustered index identify the parent rows when a table has a clustered index? How does a nonclustered index identify the data rows when a table does *not* have a clustered index?

   When a clustered index exists, the nonclustered index stores the clustered index value for each indexed row. When a clustered index does not exist, the nonclustered index stores the file ID, page number, and RID of the data row.

4. The expansion of a field that is not included in an index causes a page split. This page split has moved the row to a new page. What is the impact of this move on the nonclustered indexes in the table?

   There is no impact on the nonclustered indexes. If there were a clustered index, that clustering value would not be changed. The nonclustered index still points to the row because the clustering key has not changed. If there is no clustered index, a forward pointer is left in the position of the old record to point to the new record. In either case, the nonclustered index does not need to be modified.

5. You are considering creating a composite, clustered index on the **company name**, **last name**, and **first name** columns of a table. What are some important points to consider when you create the index, and why? Is there a better solution?

   Keep the clustered index key as small as possible. The larger the clustering key value, the greater the impact on all nonclustered indexes. The larger the clustered index, the less efficient it becomes. As the size of key values increases, the values require more space on a page; the page then holds fewer key values, which causes the clustered index tree (B-Tree) to become larger. The larger the clustered index becomes (the more non-leaf-levels it has), the more I/O cycles that are required to traverse the index tree.

   Also, this unique composite key may be better defined as a nonclustered index or as multiple indexes.

   Better solutions that you might consider are creating the clustered index on a customer ID column (if one exists) or the last name column. If a customer ID column does not exist, you should consider using the Identity property or adding a new column that contains a key value that is derived by extracting various parts of data in the row.