

Module 13: Optimizing Query Performance

Contents

Overview	1
Introduction to the Query Optimizer	2
Obtaining Execution Plan Information	13
Using an Index to Cover a Query	25
Indexing Strategies	35
Overriding the Query Optimizer	42
Recommended Practices	48
Lab A: Optimizing Query Performance	50
Review	65

Trainer Materials
for Microsoft Certified
Trainer Use Only



Information in this document is subject to change without notice. The names of companies, products, people, characters, and/or data mentioned herein are fictitious and are in no way intended to represent any real individual, company, product, or event, unless otherwise noted. Complying with all applicable copyright laws is the responsibility of the user. No part of this document may be reproduced or transmitted in any form or by any means, electronic or mechanical, for any purpose, without the express written permission of Microsoft Corporation. If, however, your only means of access is electronic, permission to print one copy is hereby granted.

Microsoft may have patents, patent applications, trademarks, copyrights, or other intellectual property rights covering subject matter in this document. Except as expressly provided in any written license agreement from Microsoft, the furnishing of this document does not give you any license to these patents, trademarks, copyrights, or other intellectual property.

© 2000 Microsoft Corporation. All rights reserved.

Microsoft, ActiveX, BackOffice, MS-DOS, PowerPoint, Visual Basic, Visual C++, Visual Studio, Windows, and Windows NT are either registered trademarks or trademarks of Microsoft Corporation in the U.S.A. and/or other countries.

Other product and company names mentioned herein may be the trademarks of their respective owners.

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:
45 Minutes

This module provides students with in-depth knowledge of how the query optimizer works to optimize queries and how to obtain execution plan information. It describes how to create indexes that cover a query, what index strategies to implement to reduce input/output (I/O), and whether to override the query optimizer.

In the lab, students will use the graphical execution plan, gather query information and view query optimizer output. They will also view index information and use that information to observe how the query optimizer optimizes queries and applies optimizer hints.

After completing this module, students will be able to:

- Explain the role of the query optimizer and how it works to ensure that queries are optimized.
- Use various methods for obtaining execution plan information so that they can determine how the query optimizer processed a query and validate that the most efficient execution plan was generated.
- Create indexes that cover queries.
- Identify indexing strategies that reduce page reads.
- Evaluate when to override the query optimizer.

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_13.ppt
- The C:\Moc\2073A\Demo\D13_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.

Module Strategy

Use the following strategy to present this module:

- **Introduction to the Query Optimizer**

Introduce the query optimizer and explain how the query optimizer takes the available information to determine the best execution plan. Then focus on the phases and details of query optimization and how execution plans are cached.

Conclude this section by briefly introducing the query governor and explain how it can be configured to prevent long-running queries from executing and from consuming system resources.

- **Obtaining Execution Plan Information**

Discuss the different ways to view the execution plan that the query optimizer generates, but primarily focus on graphically viewing the execution plan.

- **Using an Index to Cover a Query** Introduce the concept of how indexes can cover a query. Present the examples showing how the index pages can be navigated for an index that covers a query. Then, discuss the situations when the query optimizer can use an index to cover a query and how to determine whether the optimizer used an index to cover a query. Finally, provide guidelines for creating indexes that can cover queries.

- **Indexing Strategies**

Discuss specific indexing strategies for queries that retrieve ranges of data and for prioritizing multiple queries. Conclude by pointing out guidelines for creating useful indexes.

- **Overriding the Query Optimizer**

Emphasize the importance of considering other alternatives before deciding to override the query optimizer. Briefly introduce the optimizer hints and point out that optimizer hints should be tested and reevaluated periodically.

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 **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\Restore13.cmd batch file.

The lab also requires that students create the **index_cleanup** stored procedure by running the index_cleanup.sql script from C:\Moc\2073A\Labfiles\L13.

Warning If this course has been customized, students must execute the C:\Moc\2073A\Batches\Restore13.cmd batch file to ensure that the 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 how the query optimizer uses indexes and other information to determine the most efficient method of accessing data.

- Introduction to the Query Optimizer
- Obtaining Execution Plan Information
- Using an Index to Cover a Query
- Indexing Strategies
- Overriding the Query Optimizer

This module describes how the query optimizer uses indexes and other information to determine the most efficient method of accessing data.

After completing this module, you will be able to:

- Explain the role of the query optimizer and how it works to ensure that queries are optimized.
- Use various methods for obtaining execution plan information so that you can determine how the query optimizer processed a query and validate the most efficient execution plan was generated.
- Create indexes that cover queries.
- Identify indexing strategies that reduce page reads.
- Evaluate when to override the query optimizer.

◆ Introduction to the Query Optimizer

Topic Objective

To point out the topics in this section.

Lead-in

In this section, we will discuss the query optimizer and how it optimizes queries.

- Function of the Query Optimizer
- How the Query Optimizer Uses Cost-Based Optimization
- How the Query Optimizer Works
- Query Optimization Phases
- Caching the Execution Plan
- Setting a Cost Limit

Knowledge of the role of the query optimizer in optimizing queries prepares you for creating useful indexes, writing efficient queries, and tuning poorly performing queries.

Trainer Materials
for Microsoft Certified
Trainer Use Only

Function of the Query Optimizer

Topic Objective

To introduce the query optimizer.

Lead-in

The query optimizer is the component responsible for generating the optimum execution plan for a query.

- **Determines the Most Efficient Execution Plan**
 - Determining whether indexes exist and evaluating their usefulness
 - Determining which indexes or columns can be used
 - Determining how to process joins
 - Using cost-based evaluation of alternatives
 - Creating column statistics
- **Uses Additional Information**
- **Produces an Execution Plan**

The query optimizer is the component responsible for generating the optimum execution plan for a query.

Determines the Most Efficient Execution Plan

The query optimizer evaluates each Transact-SQL statement and determines the most efficient execution plan.

The query optimizer estimates the input/output (I/O) required to process a query by:

- Determining whether indexes exist and evaluating their usefulness for a query.
- Determining which indexes or columns can be used to reduce the number of rows examined by the query. By reducing the number of rows examined, the amount of I/O is reduced, which is the goal of query performance.
- Determining the most effective strategy for processing join operations, such as in which order to join tables and which join strategy to use.
- Using cost-based evaluation of alternatives to select the most efficient plan for a given query.
- Creating column statistics to improve the performance of the query.

Uses Additional Information

The query optimizer uses additional information about the underlying data and storage structures, file size, and file structure types. The query optimizer also uses an assortment from its own internal operations, such as creating temporary indexes or tables in memory, to improve the performance of queries.

Produces an Execution Plan

The query optimizer produces an execution plan that outlines the sequence of steps required to perform a query. The query optimizer optimizes the process of finding, joining, grouping, and ordering rows.

Trainer Materials
for Microsoft Certified
Trainer Use Only

How the Query Optimizer Uses Cost-Based Optimization

Topic Objective

To discuss cost-based optimization.

Lead-in

The query optimizer is a cost-based optimizer, which means that it evaluates each execution plan by estimating its execution cost.

- **Limits the Number of Optimization Plans**
 - Cost is estimated in terms of I/O and CPU cost
- **Determines Query Processing Time**
 - Use of physical operators and sequence of operations
 - Use of parallel and serial processing

The query optimizer is a cost-based optimizer, which means that it evaluates each execution plan by estimating its execution cost.

Note The cost estimates can be only as accurate as the available statistical data about the columns, indexes, and tables.

Limits the Number of Optimization Plans

To execute in a reasonable amount of time, the query optimizer limits the number of optimization plans that it considers. By evaluating sequences of the relational operations required to produce the result set, the query optimizer arrives at an execution plan that has the lowest estimated cost in terms of I/O and CPU resource loss.

Determines Query Processing Time

Query performance is determined by which physical operators the query optimizer uses and the sequence in which the operations are processed. The goal is to reduce:

- The number of rows returned.
- The number of pages read.
- The overall processing time by minimizing I/O and CPU resources used for an execution plan.

When the query optimizer optimizes queries, it does not initiate the execution plan with the lowest resource loss; it chooses the execution plan that returns results in the quickest manner to the user, with a reasonable reduction of resources.

Note If Microsoft® SQL Server™ 2000 has more than one processor available, the query optimizer may divide the query among them. Long-running queries usually benefit from parallel execution plans, but a parallel query can use more resources overall than processing a query serially.

Trainer Materials
for Microsoft Certified
Trainer Use Only

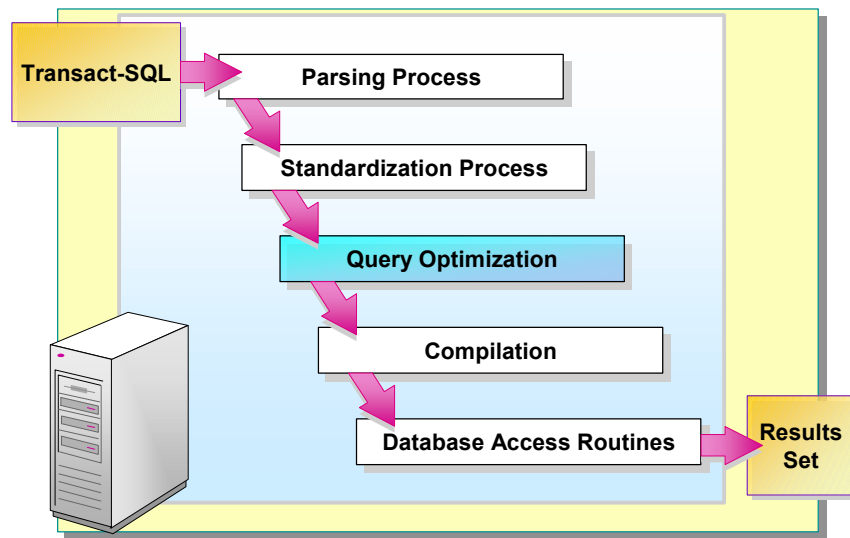
How the Query Optimizer Works

Topic Objective

To present how the query optimizer processes a query after a query is submitted to SQL Server.

Lead-in

After a query is submitted, several steps occur that transform the original query into a format that the query optimizer can interpret.

**Delivery Tip**

Explain how each step contributes to the process of transforming the original query into a format that the query optimizer can interpret.

After a query is submitted, several steps occur that transform the original query into a format that the query optimizer can interpret.

Parsing Process

The parsing process checks the incoming query for correct syntax and breaks down the syntax into component parts that the relational database engine can respond to. The output of this step is a parsed query tree.

Standardization Process

The standardization process transforms a query into a useful format for optimization. Any redundant syntax clauses that are detected are removed. Subqueries are standardized if possible. The output of this step is a standardized query tree.

Query Optimization

The process of selecting one execution plan from several possible plans is called *optimization*. Numerous steps are involved in this phase. However, the following steps have the most significant effect on the cost of the execution plan: query analysis, index selection, and join selection.

Compilation

The query is compiled into executable code.

Database Access Routines

The query optimizer determines the best method to access data, by performing a table scan, or by using an available index. The better method is then applied.

Query Optimization Phases

Topic Objective

To introduce the phases that occur during query optimization.

Lead-in

The query optimization process consists of three phases.

■ Query Analysis

- Identifies the search and join criteria of the query

■ Index Selection

- Determines whether an index or indexes exist
- Assesses the usefulness of the index or indexes

■ Join Selection

- Evaluates which join strategy to use

The query optimization process consists of three phases. These phases are not discrete processing steps and are only used to conceptually represent the internal activity of the query optimizer.

Query Analysis

The first phase of query optimization is called *query analysis*. In this phase, the query optimizer identifies the search and join criteria of the query. By limiting the search, the query optimizer minimizes the number of rows that are processed. Reducing the number of rows processed reduces the number of index and data pages read.

Index Selection

Index selection is the second phase of query optimization. During this phase, the query optimizer detects whether an index exists for the identified clauses. Then, there is an assessment of the usefulness of the index or indexes. Usefulness of an index is determined by how many rows will be returned. This information is gathered from the index statistics or column statistics. An estimate of the cost of various access methods occurs by means of estimating the logical and physical page reads required to find the qualifying rows.

Join Selection

Join selection is the third phase of query optimization. If there is a multiple-table query or self-join, there is an evaluation of which join strategy to use. The determination of which join strategy to use consists of a consideration of a number of factors: selectivity, density, and memory required to process the query.

Caching the Execution Plan

Topic Objective

To discuss how execution plans are managed in cache and how reducing plan recompilations can improve performance.

Lead-in

SQL Server has a pool of memory that is used to store execution plans and data buffers.

■ Storing a Execution Plan in Memory

- One copy for all serial executions
- Another copy for all parallel executions

■ Using an Execution Context

- An existing execution plan is reused, if one exists
- A new execution plan is generated, if one does not exist

■ Recompiling Execution Plans

- Changes in database cause execution plan to be inefficient or invalid

SQL Server has a pool of memory that is used to store execution plans and data buffers. The percentage of the pool allocated to either execution plans or data buffers fluctuates dynamically, depending on the state of the system. The part of the memory pool used to store execution plans is called the *procedure cache*.

Storing a Execution Plan in Memory

The bulk of the execution plan is a reusable, read-only data structure that can be used by any number of users. No user context is stored in the execution plan. There are never more than two copies of the execution plan in memory:

- One copy for all serial executions.
- Another copy for all parallel executions.

The parallel copy covers all parallel executions, regardless of their degree of parallelism.

Using an Execution Context

Each user executing a query has a data structure that holds the data specific to an execution, such as parameter values. This data structure is called the *execution context*. When a Transact-SQL statement is executed, SQL Server scans the procedure cache for determination of whether an execution plan exists for the same Transact-SQL statement.

- If any existing execution plan exists, SQL Server reuses the execution plan. This saves the overhead of recompiling the Transact-SQL statement.
- If no execution plan exists, SQL Server generates a new execution plan for the query.

Recompiling Execution Plans

Certain changes in a database can cause an execution plan to be either inefficient or no longer accurate. When SQL Server detects changes that invalidate an execution plan, it marks the execution plan as invalid. A new execution plan is compiled for the next connection that executes the query.

Important Performance can be improved by reducing the number of times that a plan is recompiled.

Conditions that invalidate an execution plan include:

- Any structural changes made to a table or view referenced by the query (ALTER TABLE and ALTER VIEW statements).
- New distribution statistics being generated either explicitly from a statement such as UPDATE STATISTICS, or automatically.
- Dropping an index used by the execution plan.
- An explicit call to the **sp_recompile** system stored procedure.
- Large numbers of changes to keys, or INSERT or DELETE statements for a table referenced by the query.
- For tables with triggers, if the number of rows in the inserted or deleted tables grows significantly.

Note SQL Server uses an aging algorithm to efficiently manage execution plans in cache. It evaluates cost and use of the execution plan.

Trainer Materials Certified
for Microsoft Certified
Trainer Use Only

Setting a Cost Limit

Topic Objective

To introduce the query governor.

Lead-in

You may want to control the cost of executing a query by setting a cost limit.

■ Specifying an Upper Limit

- Use the query governor to prevent long-running queries from executing and consuming system resources

■ Specifying Connection Limits

- Use the **sp_configure** stored procedure
- Execute the SET QUERY_GOVENOR_COST_LIMIT statement
- Specify 0 to turn off the query governor

You may want to control the cost of executing a query by setting a cost limit. The term *query cost* refers to the estimated elapsed time, in seconds, required to execute a query on a specific hardware configuration.

Specifying an Upper Limit

You can use the **query governor cost limit** option to prevent long-running queries from executing and consuming system resources. By default, queries are allowed to execute, no matter how long they take. The query governor uses an estimated cost to prevent queries with a high cost from executing at all.

Although the configuration value is specified in seconds, it does not truly correlate to time, but to the actual estimated cost of the query. You can specify an upper limit of the cost of the query to be executed.

Because the query governor is based on estimated query cost, rather than actual elapsed time, it does not have any run-time overhead. If the estimated cost of a query is greater than the specified cost limit, the query governor statement prevents the query from executing. This is more efficient than letting a query run until some predefined limit is reached, and then stopping the query.

Specifying Connection Limits

You can specify limits for all connections or just the queries for a specific connection. To apply query governor cost limits, you can:

- Use the **sp_configure** stored procedure to apply limits for all connections.
You can change the query governor cost limit only when **show advanced options** is set to 1. The setting takes effect immediately. You do not have to stop and restart the server.
- Execute the SET QUERY_GOVERNOR_COST_LIMIT statement to apply limits for a specific connection.
- Specify 0 (the default) to turn off the query governor. In this case, all queries are executed with no limits.

Trainer Materials
for Microsoft Certified
Trainer Use Only

◆ Obtaining Execution Plan Information

Topic Objective

To point out the topics in this section.

Lead-in

You can obtain information about the execution plan by using these methods.

- Viewing STATISTICS Statements Output
- Viewing SHOWPLAN_ALL and SHOWPLAN_TEXT Output
- Graphically Viewing the Execution Plan

The query optimizer responds to the information that it has available during the determination of the best execution plan. You can obtain information about the execution plan by querying the **sysindexes** table. You can also obtain information by using the STATISTICS statements, the SHOWPLAN statements, and graphically viewing the execution plan.

Trainer Materials
for Microsoft Certified
Trainer Use Only

Viewing STATISTICS Statements Output

Topic Objective

To discuss viewing the statistical output by using the STATISTICS statements.

Lead-in

You can use the STATISTICS IO, STATISTICS TIME, and STATISTICS PROFILE statements to get information that can help you diagnose long-running queries.

Statement	Output Sample																				
STATISTICS TIME	SQL Server Execution Times: CPU time = 0 ms, elapsed time = 2 ms.																				
STATISTICS PROFILE	<table><tr><th>Rows</th><th>Executes</th><th>StmtText</th><th>StmtId...</th></tr><tr><td>47</td><td>1</td><td>SELECT * FROM [charge] WHERE (([charge_amt]>=1)</td><td>16</td></tr><tr><td></td><td></td><td>.</td><td></td></tr><tr><td></td><td></td><td>.</td><td></td></tr><tr><td></td><td></td><td>.</td><td></td></tr></table>	Rows	Executes	StmtText	StmtId...	47	1	SELECT * FROM [charge] WHERE (([charge_amt]>=1)	16			.				.				.	
Rows	Executes	StmtText	StmtId...																		
47	1	SELECT * FROM [charge] WHERE (([charge_amt]>=1)	16																		
		.																			
		.																			
		.																			
STATISTICS IO	Table 'member'. Scan count 1, logical reads 23, physical reads 0, read-ahead reads 0.																				

You can use the STATISTICS IO, STATISTICS TIME, and STATISTICS PROFILE statements to get information that can help you diagnose long-running queries. The output from STATISTICS statements provides information about the actual execution plan.

STATISTICS TIME obtains information about the number of milliseconds required to parse, compile, and execute each statement.

STATISTICS PROFILE displays the profile information for a statement. When you execute a query, the output from the SHOWPLAN_ALL statement and two additional columns are included in the result set. The following table shows the additional columns.

Column	Description
Rows	Actual number of rows produced by each operator
Reuse	Actual number of times this operator was told to reuse its data

STATISTICS IO obtains information about the amount of page reads generated by queries. The output from STATISTICS IO includes the values in the following table.

Value	Description	Additional information
Logical reads	Number of pages read from data cache	All pages are accessed in the data cache. If a page is not available in cache, it must be physically read from disk.
Physical reads	Number of pages read from disk	<p>This value is always less than or equal to the value of logical reads.</p> <p>The following is the method for calculating the value of the Cache Hit Ratio:</p> $\text{Cache hit ratio} = \frac{\text{Logical reads} - \text{Physical reads}}{\text{Logical reads}}$
Read-ahead reads	Number of pages placed into cache	A high number for this value means that the value for physical reads is lower, and the cache hit ratio is higher than if read-ahead was not enabled.
Scan count	Number of times the table was accessed	The outer tables of a left join should always have a scan count of 1. For inner tables, the number of logical reads is determined by the scan count multiplied by the number of pages accessed on each scan.

Note The SET statements stay in effect for the session until you specify the OFF option, or until you end the session.

Trainer Material
for Microsoft Certified
Trainer Use Only

Viewing SHOWPLAN_ALL and SHOWPLAN_TEXT Output

Topic Objective

To discuss the use of the SHOWPLAN statements.

Lead-in

You can obtain detailed information about how queries are executed and how many resources are required to process the query by using the SET SHOWPLAN_TEXT and SET SHOWPLAN_ALL statements.

■ Structure of the SHOWPLAN Statement Output

- Returns information as a set of rows
- Forms a hierarchical tree
- Represents steps taken by the query optimizer
- Shows estimated values of how a query was optimized, not the actual execution plan

■ Details of the Execution Steps**■ Difference Between SHOWPLAN_TEXT and SHOWPLAN_ALL Output**

You can use the SET SHOWPLAN_TEXT and SET SHOWPLAN_ALL statements to obtain detailed information about how queries are executed and how many resources are required to process the query.

Structure of the SHOWPLAN Statement Output

The SHOWPLAN statement output:

- Returns information as a set of rows.
- Forms a hierarchical tree.
- Represents steps taken by the query optimizer to execute each statement.
- Shows estimated values of how a query was optimized, not the actual execution plan. The estimated values are based on existing statistics.

Details of the Execution Steps

Each statement reflected in the output contains a single row with the text of the statement, followed by several rows with the details of the execution steps.

Details of the execution steps include:

- Which indexes are used with which tables.
- The join order of the tables.
- The chosen update mode.
- Worktables and other strategies.

Difference Between SHOWPLAN_TEXT and SHOWPLAN_ALL Output

The difference between SHOWPLAN_TEXT and SHOWPLAN_ALL output is that the SHOWPLAN_ALL output returns additional information, such as the estimated rows, I/O, CPU, and average row size of the query.

Note The SET statements stay in effect for the session until you specify the OFF option, or until you end the session.

Trainer Materials
for Microsoft Certified
Trainer Use Only

◆ Graphically Viewing the Execution Plan

Topic Objective

To point out the topics in this subsection.

Lead-in

You can use SQL Query Analyzer to graphically view a color-coded execution plan.

- Elements of the Graphical Execution Plan
- Reading the Graphical Execution Plan Output
- Using the Bookmark Lookup Operation

You can use SQL Query Analyzer to graphically view a color-coded execution plan.

Trainer Materials
for Microsoft Certified
Trainer Use Only

Elements of the Graphical Execution Plan

Topic Objective

To introduce elements of the graphical execution plan.

Lead-in

The graphical execution plan, which contains the following elements, uses icons to represent the execution of specific parts of statements and queries.

- **Steps Are Units of Work to Process a Query**
- **Sequence of Steps Is the Order in Which the Steps Are Processed**
- **Logical Operators Describe Relational Algebraic Operation Used to Process a Statement**
- **Physical Operators Describe Physical Implementation Algorithm Used to Process a Statement**

Delivery Tip

Briefly review the icon, physical operator, and description that the query optimizer uses. A list of physical operators follows.

The graphical execution plan, which contains the following elements, uses icons to represent the execution of specific parts of statements and queries:










Steps are units of work used to process a query.

Sequence of steps is the order in which the steps are processed.

Logical operators describe the relational algebraic operation used to process a statement; for example, performing an aggregation. The logical operator typically matches the physical operator. Not all steps used to process a query or update operations involve logical operations.

Physical operators describe the physical implementation algorithm used to process a statement; for example, scanning a clustered index. Each step in the execution of a query or update operation involves a physical operator.

The following table is a partial list of physical operators used to represent the algorithms that the query optimizer uses.

Icon	Physical operator	Operator description
	Bookmark Lookup	Uses a bookmark (row ID or clustering key) to look up the corresponding row in the table or clustered index
	Filter	Scans the input, returning only those rows that satisfy the filter expression that appears in the argument column
	Hash Match	Builds a hash table by computing a hash value for each row from its build input
	Index Scan	Retrieves all rows from the nonclustered index specified in the argument column
	Index Seek	Uses the seeking ability of indexes to retrieve rows from a nonclustered index
	Merge Join	Performs all types of joins (except self-join and cross join), including UNION operations
	Nested Loops	Searches the inner table for each row of the outer table, typically by using an index
	Sort	Sorts all incoming rows
	Table Scan	Retrieves all rows from the table specified in the argument column

Note For the complete listing of icons and more information, search on “graphically displaying the execution plan using SQL Query Analyzer” in SQL Server Books Online.

Trainer Material
for Microsoft Certified
Trainer Use Only

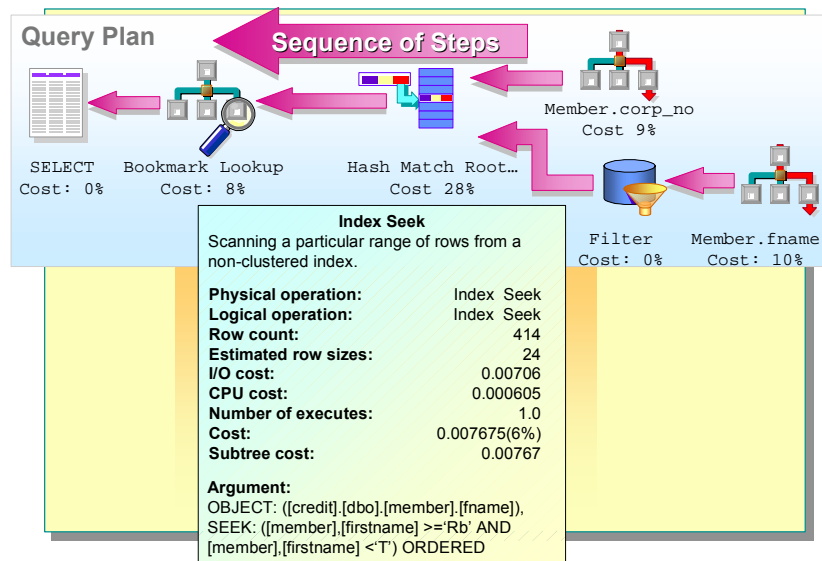
Reading Graphical Execution Plan Output

Topic Objective

To discuss how to read the execution plan output.

Lead-in

The graphical execution plan output is read from right to left and from top to bottom. Each query in the batch that is analyzed is displayed; this includes the cost of each query as a percentage of the total cost of the batch.



Delivery Tip

Using SQL Query Analyzer, turn on Show Execution Plan, and then execute a query.

In the Execution Plan output, place the pointer on an icon to display the additional information about that particular operation.

The graphical execution plan output is read from right to left and from top to bottom. Each query in the batch that is analyzed is displayed, including the cost of each query as a percentage of the total cost of the batch.

Each step can have one or many nodes to process. The term *node* refers to an operation that the query optimizer uses, which is represented by an icon.

The execution plan may have multiple nodes for a particular step.

- Each node is related to a parent node.
- All nodes with the same parent are drawn in the same column.
- Arrowheads connect each node to its parent.
- Recursive operations are shown with an iteration symbol.
- Operators are shown as symbols related to a specific parent.
- When the batch contains multiple statements, multiple execution plans are drawn.

Delivery Tip

Briefly describe the types of detailed information that you can view.

Viewing Additional Information

When you place the pointer on each node (represented by an icon), you can view detailed information about the physical and logical operators, in addition to the information in the following table.

Measures	Description
Row count	The number of rows returned by the operator.
Estimated row size	The estimated size of the row returned by the operator.
I/O cost	The estimated cost of all I/O activity for the operation. This value should be as low as possible.
CPU cost	The estimated cost for all CPU activity for the operation.
Number of executes	The number of times that the operation was executed during the query.
Cost	The cost to the query optimizer when executing the operation, including cost of this operation as a percentage of the total cost of the query.
Subtree cost	The total cost to the query optimizer when executing this operation and all operations preceding it in the same subtree.
Argument	The predicates and parameters used by the query.

Trainer Materials
for Microsoft Certified
Trainer Use Only

Using the Bookmark Lookup Operation

Topic Objective

To describe how the query optimizer uses the Bookmark Lookup operation.

Lead-in

Bookmark Lookup is an internal operator used by the query optimizer.

- **Analyzing the Query Plan**
 - Typically used after all steps have been processed
- **Retrieving Rows**
 - Row identifiers
 - Clustering Keys
- **Observing the Details**
 - A bookmark label used to find the row
- **Determining When the Bookmark Lookup Operator is Used**
 - Queries containing the IN clause or the OR operator

Bookmark Lookup is an internal operator frequently used by the query optimizer. When the query optimizer identifies records that are possible candidates for the intended result set, it notes the information identifying the row locations (a bookmark) and continues operations that refine the search.

If a row is included in the search, SQL Server uses the row location from the bookmark to find the row by analyzing the query plan, retrieving rows, observing the details, and determining when the Bookmark Lookup operator is used.

Analyzing the Query Plan

In the query plan, the query optimizer typically uses the Bookmark Lookup operator after all other steps have been processed.

Retrieving Rows

The Bookmark Lookup operator retrieves all the qualifying rows by using:

- A row identifier (RID) to find the corresponding row in a heap.
- The clustering key to find the corresponding row in a clustered index.

Observing the Details

In the query plan, details of the Bookmark Lookup operator contain:

- A bookmark label used to find the row in the table or clustered index.
- The table name or clustered index name from which the row is found.
- The WITH PREFETCH clause, if the query optimizer determines that read-ahead is the best way to find bookmarks in the table or clustered index.

Determining When the Bookmark Lookup Operator Is Used

The query optimizer typically uses the Bookmark Lookup operator to process queries containing the IN clause and OR operators in the WHERE clause.

Example

In this example, the **member** table has a nonclustered index on the **member_no** column. The query optimizer uses a Bookmark Lookup operator to retrieve the qualifying rows.

```
USE credit
SELECT *
FROM member
WHERE member_no
IN (4567,8765,4321)
```

Trainer Materials
for Microsoft Certified
Trainer Use Only

◆ Using an Index to Cover a Query

Topic Objective

To point out the topics in this section.

Lead-in

You can create indexes that satisfy the query without having to access the data pages.

- Introduction to Indexes That Cover a Query
- Locating Data by Using Indexes That Cover a Query
- Identifying Whether an Index Can Be Used to Cover a Query
- Determining Whether an Index Is Used to Cover a Query
- Guidelines for Creating Indexes That Cover a Query

You can create indexes that resolve the query without having to access the data pages. This is a strategy that can improve query performance.

Trainer Materials
for Microsoft Certified
Trainer Use Only

Introduction to Indexes That Cover a Query

Topic Objective

To introduce the concept of indexes covering a query.

Lead-in

When creating indexes, you may want to create an index that covers the most common queries in order to reduce the amount of I/O.

- **Only Nonclustered Indexes Cover Queries**
- **Indexes Must Contain All Columns Referenced in the Query**
- **No Data Page Access Is Required**
- **Indexed Views Can Pre-Aggregate Data**
- **Indexes That Cover Queries Retrieve Data Quickly**

When creating indexes, you may want to create an index that covers the most common queries in order to reduce the amount of I/O.

Only Nonclustered Indexes Cover Queries

Indexes that cover queries contain all of the required data of a query in the leaf level of a nonclustered index.

Indexes Must Contain All Columns Referenced in the Query

An index that covers a query must contain all columns that are referenced in the SELECT statement. If a clustered index exists, the fields in the clustering key are in the leaf level of the nonclustered index and contribute to covering the query.

No Data Page Access Is Required

When a query is covered by an index, the query optimizer does not access the data pages, because all of the required data is contained in the index. The amount of I/O is significantly reduced.

Indexed Views Can Pre-Aggregate Data

If an indexed view sums, counts, or averages columns, then the query optimizer can use this view to provide stored values when resolving a query. Indexed views that pre-aggregate data can increase performance dramatically.

Indexes That Cover Queries Retrieve Data Quickly

Creating indexes that cover queries is one of the fastest ways to access to data, especially for a low-selectivity query. When you compare the leaf levels of the clustered and nonclustered indexes, the advantage of having indexes that cover queries are evident.

Index type	Contents of leaf level
Clustered	Entire row (actual data pages)
Nonclustered	Key value

Because key values are typically smaller in size than the actual rows, an index page can store more key values than complete rows. Storing key values requires fewer pages, which reduces the amount of I/O.

Trainer Materials
for Microsoft Certified
Trainer Use Only

◆ Locating Data by Using Indexes That Cover a Query

Topic Objective

To discuss the topics in this subsection.

Lead-in

The query optimizer navigates the leaf level in different ways when an index can be used to cover a query.

- Example of Single Page Navigation
- Example of Partial Scan Navigation
- Example of Full Scan Navigation

The query optimizer navigates the leaf level in different ways when an index can be used to cover a query. Covering a query can consist of reading a single page, a range of pages, or all of the pages of the leaf level. The data pages are never accessed.

Trainer Materials
for Microsoft Certified
Trainer Use Only

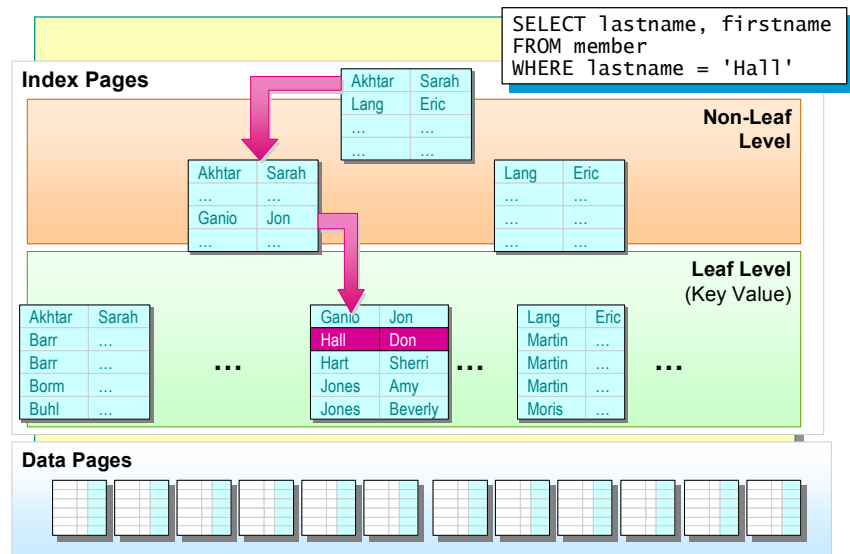
Example of Single Page Navigation

Topic Objective

To illustrate single page navigation of an index.

Lead-in

In this example, a query that is covered by an index requires reading a single leaf-level page.



Single page navigation occurs when only one page of the leaf-level pages is read from the non-leaf-level. Reading one page is similar a point query, where the information—a single row or multiple rows—is found on a single page.

Note Single page navigation does not mean that the query can return only one row. A point query can return one row or all of the rows on one page. Either way, all of the data is found on one page.

Example

In this example, a composite, nonclustered index on the **lastname**, **firstname** columns covers the query.

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

SQL Server goes through the following steps to retrieve the information:

1. Traverses the index tree comparing the last name Hall to the key values.
2. Continues to traverse the index until it reaches the first page of the leaf level containing the key value Hall.
3. Returns the qualifying rows without accessing the data pages, because the **lastname** and **firstname** key values are contained in the leaf level.

For More Information

For simplicity, the pointer from the leaf level of the nonclustered index to the data pages (heap or clustered index) is not shown on the slide.

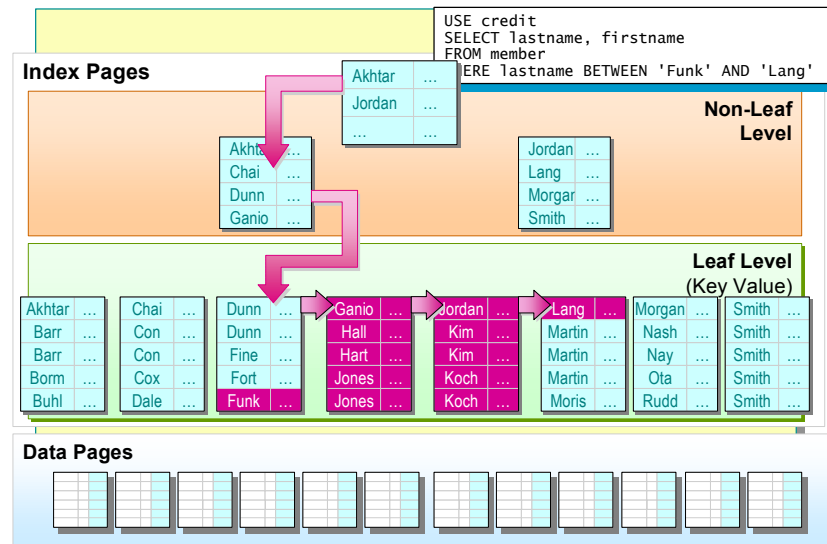
Example of Partial Scan Navigation

Topic Objective

To illustrate partial scan navigation of an index.

Lead-in

In this example, a query that is covered by an index requires reading a number of leaf-level pages.



A partial scan occurs when a range of pages is read from the leaf level.

Example

In this example, a composite, nonclustered index on the **lastname, firstname** columns covers the query by doing a partial scan of the leaf-level pages.

```
USE credit
SELECT lastname, firstname
FROM member
WHERE lastname BETWEEN 'Funk' AND 'Lang'
```

For Your Information

The **firstname** column is omitted to simplify the slide. Refer to the previous slide if this slide is unclear.

SQL Server goes through the following steps to retrieve the information:

1. Traverses the index tree.
2. Starts reading leaf-level pages at the page that contains the first occurrence of the last name Funk.
Data in the leaf level is sorted in ascending order.
3. Reads the range of leaf-level pages through to the last name of Lang.
At this time, the partial scan is completed.
4. Returns the qualifying rows without accessing the data pages, because the leaf level is scanned for last names between Funk and Lang.

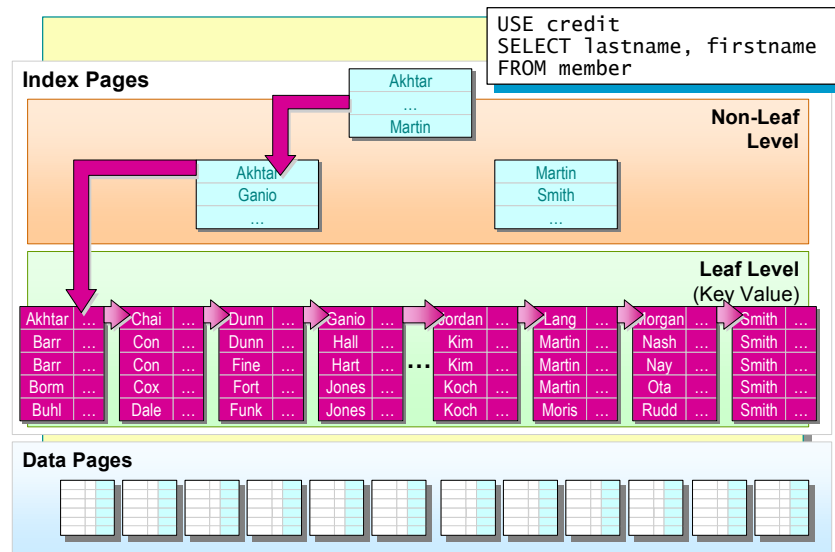
Example of Full Scan Navigation

Topic Objective

To illustrate full scan navigation of an index.

Lead-in

In this example, a query that is covered by an index requires reading all of the leaf-level pages.



A full scan occurs when all of the pages of the leaf level are read. Similar to a table scan, a full scan occurs when a query does not include a WHERE clause, or when the WHERE clause is not selective.

Example

In this example, a composite, nonclustered index on the **lastname**, **firstname** columns covers the query by doing a full scan of the leaf-level pages.

```
USE credit
SELECT lastname, firstname
FROM member
```

To retrieve the information, SQL Server:

1. Traverses the index tree.
2. Reads the leaf-level pages, starting with the first page, and scans through all of the leaf-level pages until it reaches the last page in the leaf-level.
3. Returns the qualifying rows without accessing the data pages because the leaf-level is scanned.

For Your Information

This slide illustrates a scan for data ordered by **lastname**. If data is requested, unordered or ordered by **firstname**, then SQL Server may use the allocation pages to identify and scan all index pages, and then throw out the non-leaf pages. The **firstname** column is omitted to simplify the slide.

Note Scanning the leaf level of an index also is a parallel data scan. SQL Server uses read-ahead processing to further improve the performance of the query.

Identifying Whether an Index Can Be Used to Cover a Query

Topic Objective

To point out when the query optimizer can use an index to cover a query.

Lead-in

These factors affect the ability of an index to cover a query.

- **All Necessary Data Must Be in the Index**
- **A Composite Index Is Useful Even if the First Column Is Not Referenced**
- **A WHERE Is Not Necessary**
- **A Nonclustered Index Can Be Used if It Requires Less I/O Than a Clustered Index Containing a Column Referenced in the WHERE Clause**
- **Indexes Can Be Joined to Cover a Query**

These factors affect the ability of an index to cover a query:

- All necessary data must be in the index. This data includes all referenced columns, whether they are returned in the result set, used for sorting or aggregation, or supplied in the WHERE clause.
- A column in an index can contribute to covering a query even when it is not the first column referenced in a composite index.
For example, a composite index on **SalesRep**, **Region**, **Amount** (in that order) could cover a query that referenced only **Region** and **SUM(Amount)**.
- A WHERE clause is not necessary. The query optimizer scans the entire leaf level.
- A nonclustered index can be used to cover a query if it requires less I/O than a clustered index containing a column referenced in the WHERE clause.
- Indexes can be joined to cover a query. If some or all tables referenced in a join operation have an index that covers a query, the results are joined together by a special join operation, and the rows are then returned.

Determining Whether an Index Is Used to Cover a Query

Topic Objective

To point out how to determine whether a query is covered by an index.

Lead-in

Queries that are covered by an index are not explicitly apparent to users. You can observe the graphical execution plan or compare I/O to determine whether the query optimizer used an index to cover a query.

■ Observing the Execution Plan Output

- Displays the phrase “Scanning a non-clustered index entirely or only a range”

■ Comparing I/O

- Nonclustered index
 - Total number of levels in the non-leaf level
 - Total number of pages that make up the leaf level
 - Total number of rows per leaf-level page
 - Total number of rows per data page
- Total number of pages that make up the table

Queries that are covered by an index are not explicitly apparent to users. You can observe the graphical execution plan or compare I/O to determine whether the query optimizer used an index to cover a query.

Observing the Execution Plan Output

You can view the execution plan graphically. If an execution plan output displays the phrase “Scanning a non-clustered index entirely or only a range,” the query optimizer was able to cover the query by using an index.

Comparing I/O

You can also view the STATISTICS IO output. When evaluating the cost of an index that covers a query, remember that the query optimizer always attempts to cover the query when evaluating an execution plan.

To help you determine whether the query is covered, you should know the following information about the nonclustered index and table:

- Nonclustered index
 - Total number of levels in the non-leaf level
 - Total number of pages that make up the leaf level
 - Total number of rows per leaf-level page
 - Total number of rows per data page
- Total number of pages that make up the table

Note If you prefer, you can also calculate the size of the leaf level of a nonclustered index rather than using the STATISTICS IO statement. Alternately, you can query **sysindexes** and review the **dpages** column, which will display the size of the leaf level.

Guidelines for Creating Indexes That Cover a Query

Topic Objective

To discuss some guidelines when creating indexes that can cover queries.

Lead-in

When creating indexes that cover queries, consider these guidelines.

- **Add Columns to Indexes**
- **Minimize Index Key Size**
- **Maintain Row-to-Key Size Ratio**

When creating indexes that cover a query, consider the following guidelines:

- Add columns to indexes. You may want to add columns to some indexes that:
 - Cover more than one query.
 - Contribute toward covering some of your more common queries.
 - Are referenced frequently.
 - Do not significantly add to the key size.
- Minimize index key size. When defining the index key (key values), avoid specifying key values that are too wide. Wide rows increase row size, the number of index levels, and the total number of pages. Any performance benefits gained from creating an index that covers queries would be reduced.
- Maintain row-to-key size ratio. If the size of the index key increases relative to the row size, query performance may be affected. An extreme example is if you created a nonclustered index on all of the columns in a table. By doing this, a virtual copy of the table is produced and stored in the leaf level of the nonclustered index in sorted order.

Delivery Tip

Mention that columns that may be too large in one instance might be acceptable in another.

Ask students how many characters are too many for a table that they want to optimize.

◆ Indexing Strategies

Topic Objective

To point out the topics in this section.

Lead-in

You can implement indexing strategies to improve query performance.

- Evaluating I/O for Queries That Access a Range of Data
- Indexing for Multiple Queries
- Guidelines for Creating Indexes

You can implement indexing strategies to improve query performance.

Trainer Materials
for Microsoft Certified
Trainer Use Only

Evaluating I/O for Queries That Access a Range of Data

Topic Objective

To illustrate the differences across page I/O by using different access methods.

Lead-in

The query optimizer automatically considers multiple execution plans and estimates the needed I/O for each execution plan.

```
SELECT charge_no
FROM charge
WHERE charge_amt BETWEEN 20 AND 30
```

<i>Access method</i>	<i>Page I/O</i>
Table scan	10,417
Clustered index on the charge_amt column	1042
Nonclustered index on the charge_amt column	100,273
Composite index on charge_amt, charge_no columns	273

Delivery Tip

The example cannot be executed against the **credit** database.

Refer to the assumptions in the workbook when comparing page I/O for the different indexes.

The query optimizer automatically considers multiple execution plans and estimates the needed I/O for each execution plan. It then initiates an execution plan with the least amount of I/O in addition to other considerations. Compare the page I/O among the different access methods that the query optimizer can use.

For example, consider the following query that retrieves a range of data, and then compare the I/O of this query against different methods of accessing data.

```
SELECT charge_no
FROM charge
WHERE charge_amt BETWEEN 20 AND 30
```

Assume the following when comparing the different methods:

- There are 1 million rows, and 96 rows per page.
- The total number of pages is 10,147.
- There is no clustered index.
- 100,000 rows fall within the \$20.00 to \$30.00 range.
- 367 index rows fit on a nonclustered index leaf page.

Delivery Tip

Point out that the access methods illustrated on the slide use this information.

Table Scan

Performing a table scan is advantageous for queries where the result set includes a high percentage of a table (low selectivity). Table scans are appropriate when the total page I/O of a query would exceed the number of pages in the table.

When you execute the query that does a table scan, the page I/O is 10,417. Compare the page I/O on a table scan to a nonclustered index on the **charge_amt** column. Performing a table scan is more efficient.

Clustered Index on the charge_amt Column

SQL Server performs the following steps to retrieve the information:

1. Searches clustered index for the minimum value, in this case \$20.00.
2. Reads rows starting at \$20.00 and stops the search at \$30.00.

Because the **charge_amt** column is clustered, the physical order of the data is arranged according to charge amount. All of the data that falls within that range is in sequential order on subsequent pages, making it easy to retrieve data. The page I/O is 1,042 (100,000/96 rows per page).

Nonclustered Index on the charge_amt Column

SQL Server goes through the following steps to retrieve the information:

1. Searches for the range of values in the leaf level of the nonclustered index and retrieves the RID for each row. In this case, 273 leaf-level pages are accessed (100,000/367).
2. Data is retrieved from each page by using the Bookmark Lookup for each qualifying row.

The page I/O is approximately 100,273. To retrieve data by using a nonclustered index on the **charge_amt** column is the least effective method, because SQL Server must read one page for every row—*plus* the leaf level of the index is read to retrieve the RID values. Each data page is read multiple times in cache.

Composite Index on the charge_amt, charge_no Columns

The page I/O is 273 (100,000/367 rows per page). The number of index rows per leaf level averages 367. Because the **charge_amt** and **charge_no** columns are in the index, SQL Server does not search the data pages, which reduces the amount of I/O.

Trainer Material
for Microsoft
Trainer Use

Indexing for Multiple Queries

Topic Objective

To illustrate the challenge of creating indexes to support the most important queries.

Lead-in

Choosing the most appropriate index to create based on an individual query is easier than creating an index for multiple-priority queries.

Example 1

```
USE credit
SELECT charge_no, charge_dt, charge_amt
FROM charge
WHERE statement_no = 19000 AND member_no = 3852
```

Example 2

```
USE credit
SELECT member_no, charge_no, charge_amt
FROM charge
WHERE charge_dt between '07/30/1999'
AND '07/31/1999' AND member_no = 9331
```

Delivery Tip

Use the examples on the slide and refer to the table in the student workbook when comparing query performance for both queries against different indexing strategies.

To choose the most appropriate index to create, based on an individual query is easier than creating an index for multiple-priority queries. To create indexes to support multiple-priority queries is more complex because the best index for one query may not be the best index for another. The goal is to attain acceptable performance for all of the highest-priority queries by evaluating I/O.

Example Business Scenario

For the following examples, assume that the most common queries requested by users are finding customer charges for a specific statement (Example 1) and finding customer charges for a specific day (Example 2). The first example query is 15 percent of the table. The other query is highly selective, accessing only a few rows.

Example 1

```
USE credit
SELECT charge_no, charge_dt, charge_amt
FROM charge
WHERE statement_no = 19000 AND member_no = 3852
```

Example 2

```
USE credit
SELECT member_no, charge_no, charge_amt
FROM charge
WHERE charge_dt between '07/30/1999' AND '07/31/1999'
AND member_no = 9331
```

The following table compares the query performance of Examples 1 and 2, based on the possible indexing strategy that you may implement. A clustered index on the **member_no** column is the best strategy.

Type of index	Column	Example 1 query	Example 2 query
Clustered	member_no	Very fast.	Very fast.
Nonclustered	charge_no	Uses the clustered index.	Uses the clustered index.
Clustered	charge_no	Slower than if a clustered index were created on the member_no column.	Slow. The nonclustered index on member_no is not efficient with ranges of data.
Nonclustered	member_no		
Clustered	member_no	Very fast.	Very fast.
Nonclustered, composite	statement_no, member_no	Uses the clustered index.	Uses the clustered index.
Clustered	charge_no	Slower than if a clustered index were created on the member_no column.	Fast.
Nonclustered, composite	member_no, charge_dt		A composite index significantly increases performance of the nonclustered index.

Trainer Materials
for Microsoft Certified
Trainer Use Only

Guidelines for Creating Indexes

Topic Objective

To present guidelines for creating useful indexes.

Lead-in

To ensure that the indexes that you create are useful to the query optimizer, consider the following guidelines.

- **Determine the Priorities of All of the Queries**
- **Determine the Selectivity for Each Portion of the WHERE Clause of Each Query**
- **Determine Whether to Create an Index**
- **Identify the Columns That Should Be Indexed**
- **Determine the Best Column Order of Composite Indexes**
- **Determine What Other Indexes Are Necessary**
- **Test the Performance of the Queries**

Your decision on how many indexes, the type of indexes, and the columns on which to create indexes should be based on a thorough understanding of the data and the needs of users.

To ensure that the indexes that you create are useful to the query optimizer, consider the following guidelines:

- Determine the priorities of all of the queries.
 - Gain a thorough understanding of the data and how it will be used.
 - Determine the priority transactions for the database.
- Determine the selectivity for each portion of the WHERE clause of each query.
- Determine whether to create an index.

There are situations when you will not want to create an index. These include:

- If the index is never used by the query optimizer.
- If the column values are low in selectivity.
- If the column to be indexed is too wide.
- Identify the columns that should be indexed.
 - Create an index on a column that is used as a join key to improve the performance of the join. This allows the query optimizer the option to use an index rather than perform a table scan.
 - Evaluate whether the column is searched frequently.
 - Ensure that the columns referenced in the WHERE clauses of the highest-priority queries are indexed.
- Determine the best column order of composite indexes.

- Determine what other indexes are necessary.
 - Determine the minimum number of indexes that can be created for each table.
 - Balance the performance gain of the index versus the update maintenance.
 - If a query is executed infrequently, you may want to consider creating an index for the duration of a specific activity (when it can provide a significant performance gain) and then dropping it.
- Test the performance of the queries.

After the indexes are created, test the performance of the highest-priority queries by executing the following statements for each query:

- SET SHOWPLAN ON
- SET STATISTICS IO ON
- SET STATISTICS TIME ON

Trainer Materials
for Microsoft Certified
Trainer Use Only

◆ Overriding the Query Optimizer

Topic Objective

To point out the topics in this section.

Lead-in

This section discusses ways to override the query optimizer.

- **Determining When to Override the Query Optimizer**
- **Using Hints and SET FORCEPLAN Statement**
- **Confirming Query Performance After Overriding the Query Optimizer**

This section discusses ways to override the query optimizer and how to determine when to do it. When you do override the query optimizer, it is important to test and reconfirm query performance.

Trainer Materials
for Microsoft Certified
Trainer Use Only

Determining When to Override the Query Optimizer

Topic Objective

To point out alternatives to overriding the query optimizer.

Lead-in

It is usually not a good idea to override the query optimizer.

- **Limit Optimizer Hints**
- **Explore Other Alternatives Before Overriding the Query Optimizer by:**
 - Updating statistics
 - Recompiling stored procedures
 - Reviewing the queries or search arguments
 - Evaluating the possibility of building different indexes

If queries do not perform efficiently, you may choose to override the query optimizer by using *optimizer hints*. Optimizer hints are keywords that you include in your query to force a specific optimization operation.

You should limit the use of optimizer hints because they force optimization to become static. Optimizer hints prevent the query optimizer from adjusting to a changing environment. After you use optimizer hints, you must constantly monitor query performance to verify that the query performs optimally.

Before you consider overriding the query optimizer, you should explore all other alternatives by:

- Updating statistics.
- Recompiling stored procedures.
- Reviewing the queries or search arguments to determine whether you should rewrite them.
- Evaluating the possibility of building different indexes.

Using Hints and SET FORCEPLAN Statement

Topic Objective

To discuss overriding the query optimizer.

Lead-in

You can override the query optimizer by using optimizer hints or the SET FORCEPLAN statement.

- Table Hints
- Join Hints
- Query Hints
- SET FORCEPLAN Statement

You can override the query optimizer by using hints or the SET FORCEPLAN statement. You can specify a query optimizer hint within SELECT, INSERT, UPDATE, or DELETE statements. There are three types of hints that can be used for overriding the query optimizer.

Table Hints

A *table hint* specifies a table scan, one or more indexes to be used by the query optimizer, or a locking method to be used by the query optimizer with this table and for a statement. When using the table hints, consider the following:

- Each table hint can be specified only once, although you can have multiple table hints
- The WITH clause must be specified next to the table name

Syntax

```
table_name [ [ AS ] table_alias ] [ WITH ( < table_hint > [ ,...n ] ) ]
```

```
WITH ( < table_hint > ) ::=
{ INDEX ( index_val [ ,...n ] )
| FASTFIRSTROW
| HOLDLOCK
| NOLOCK
| PAGLOCK
| READCOMMITTED
| READPAST
| READUNCOMMITTED
| REPEATABLEREAD
| ROWLOCK
| SERIALIZABLE
| TABLOCK
| TABLOCKX
| UPDLOCK
| XLOCK
}
```

Join Hints

Join hints enforce a join strategy between two tables. Join hints are specified in a query's FROM clause. When a join hint is specified, the query optimizer automatically enforces the join order for all joined tables in the query, based on the position of the ON keywords.

Syntax

```
< join_hint > ::=
    { LOOP | HASH | MERGE | REMOTE }
```

Query Hints

Delivery Tip

Remind students that the UNION operator increases the number of rows, whereas join operations increase the number of columns.

Query hints can control a wide variety of actions. You can specify the query optimizer to use a particular hint for a query by using the OPTION clause. When using the OPTION clause, consider the following facts:

- Each query hint can be specified only once, although you can have multiple query hints.
- The OPTION clause must be specified with the outermost query of the statement.
- The query hint affects all operators in the statement.
- If a UNION is involved in the main query, only the last query involving a UNION operator can have the OPTION clause.

Syntax

```
[ OPTION ( < query_hint > [ ,...n ] )
  < query_hint > ::=
    { { HASH | ORDER } GROUP
    | { CONCAT | HASH | MERGE } UNION
    | { LOOP | MERGE | HASH } JOIN
    | FAST number_rows
    | FORCE ORDER
    | MAXDOP number
    | ROBUST PLAN
    | KEEP PLAN
    | KEEPFIXED PLAN
    | EXPAND VIEWS
    }
```

SET FORCEPLAN Statement

By using the FROM clause, you can force the query optimizer to join tables in the order in which they are listed. When using the SET FORCEPLAN statement, the query optimizer uses nested loop joins only.

The SET FORCEPLAN statement is a session-level setting.

Syntax

```
SET FORCEPLAN {ON | OFF}
```

Note If one or more query hints cause the query optimizer to not generate a valid execution plan, SQL Server cancels the execution and issues error message 8622. You must resubmit the query without specifying any optimizer hints or using the SET FORCEPLAN statement.

Trainer Materials
for Microsoft Certified
Trainer Use Only

Confirming Query Performance After Overriding the Query Optimizer

Topic Objective

To discuss the importance of testing and reevaluating query performance *after* overriding the query optimizer.

Lead-in

If you determine that overriding the query optimizer is necessary, test and reevaluate query performance.

- **Verify That Performance Improves**
- **Document Reasons for Using Optimizer Hints**
- **Retest Queries Regularly**

If you determine that overriding the query optimizer is necessary, verify that performance has improved, document your reasons for overriding the query optimizer, and retest the queries regularly.

Verify That Performance Improves

To verify that the query optimizer hints will improve performance, specify the ON option for the STATISTICS IO and STATISTICS TIME statements and select **Show Execution Plan in Query Analyzer**. In most cases, overriding the query optimizer does not improve performance.

If you are passing input values in a stored procedure, verify that performance is not compromised for *any* of the inputs. Optimizer hints can improve performance for certain input values, but may compromise performance for other input values.

Document Reasons for Using Optimizer Hints

If overriding the query optimizer improves performance, be sure that you document the reasons why. To document your reasons allows you to periodically reevaluate the validity of the optimizer hints. If those reasons change, the optimizer hints may no longer be necessary and may compromise performance.

Retest Queries Regularly

The query optimizer is dynamic and is constantly evaluating the best execution plan as your data changes. If you use optimizer hints, the execution plan becomes static. For this reason, you should consider retesting, on a regular basis, any queries for which you override the query optimizer.






Recommended Practices

Topic Objective

To list the recommended practices for improving query performance.

Lead-in

These recommended practices can help you optimize query performance.

-  **Use the Query Governor to Prevent Long-Running Queries from Consuming System Resources**
-  **Have a Thorough Understanding of the Data and How Queries Gain Access to Data**
-  **Create Indexes That Cover the Most Frequently Used Queries**
-  **Establish Indexing Strategies for Individual and Multiple Queries**
-  **Avoid Overriding the Query Optimizer**

These recommended practices will help you with indexing strategies that can ensure or improve query performance.

- Use the query governor to prevent long-running queries from executing and consuming system resources. By default, queries are allowed to execute, no matter how long they take. The query governor uses an estimated cost to prevent queries with high cost from executing at all.
- Have a thorough understanding of the data and how queries access it. To know how the query optimizer moves through clustered indexes, nonclustered indexes, and indexes that cover queries, enables design of effective indexes for the queries that your users execute.
- Create indexes that cover the most frequently used queries. When a query is covered by an index, the query optimizer does not access the data pages, because all of the required data is contained in the index. The amount of I/O is significantly reduced.
- Establish indexing strategies for individual and multiple queries. Strive to attain acceptable performance for each of the high-priority queries.
- Avoid overriding the query optimizer. The query optimizer generally selects the most efficient execution plan. If you use optimizer hints, they may become outdated and negatively affect query performance.

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

Topic	Search on
Graphically displaying the execution plan using icons	“graphically displaying the execution plan using SQL Query Analyzer”
Caching the execution plan	“execution plan caching and reuse”
Creating indexes that cover a query	“using indexes on views”, “resolving indexes on views”

Trainer Materials
for Microsoft Certified
Trainer Use Only

Lab A: Optimizing Query Performance

Topic Objective

To introduce the lab.

Lead-in

In this lab, you will create indexes and analyze query performance.



Explain the lab objectives.

Objectives

After completing this lab, you will be able to:

- Use the graphical execution plan to determine how a query is resolved.
- Compare I/O for queries that are covered or not covered by indexes.
- Compare I/O for queries that retrieve a range of data.
- Use optimizer hints to force the use of an index and join method.

Prerequisites

Before working on this lab, you must have:

- Script files for this lab, which are located in C:\Moc\2073A\Labfiles\L13.
- Answer files for this lab, which are located in C:\Moc\2073A\Labfiles\L13\Answers.

Lab Setup

To complete this lab, you must have either:

- Completed the prior lab, or
- Executed the C:\Moc\2073A\Batches\Restore13.cmd batch file.

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

- Created the **index_cleanup** stored procedure by running Index_cleanup.sql, which is located in C:\Moc\2073A\Labfiles\L13.

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 **credit** database schema.
- 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: 45 minutes

Exercise 1

Use the Graphical Execution Plan to Determine How a Query Is Resolved

In this exercise, you will create an index on a computed column and use the graphical execution plan to determine whether the index is useful.

You can open, review, and execute sections of the `Indexed_View.sql` script file in `C:\Moc\SQL2073A\Labfiles\L13`, or type and execute the provided Transact-SQL statements.

► To create an indexed view

In this procedure, you will drop all existing indexes on the **charge** table in the **credit** database and create an indexed view that summarized charges by member.

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

Option	Value
User name	SQLAdminx (where <i>x</i> 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 Microsoft SQL Server 2000 because you are logged as **SQLAdminx**, 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. Type and execute this statement to drop existing indexes on the **charge** table in the **credit** database:

```
USE credit
EXEC index_cleanup charge
```

4. Type and execute this statement to create a view on the **charge** table in the **credit** database:

```
CREATE VIEW mem_charges
WITH SCHEMABINDING
AS
SELECT member_no, SUM(charge_amt) AS charge_SUM,
COUNT_BIG(*) AS mem_count
FROM dbo.charge GROUP BY member_no
```

5. Type and execute these two statements to create indexes on the **mem_charges** view:

```
CREATE UNIQUE CLUSTERED INDEX c1_mem_chg ON
mem_charges(member_no)
CREATE NONCLUSTERED INDEX nc_mem_chg_amt ON
mem_charges(charge_SUM)
```

► **To view the graphical execution plan.**

In this procedure, you will query the **charge** table and view the query execution plan to determine how the query optimizer obtained the query result.

1. In the Query window, on the **Query** menu, click **Show Execution Plan** to start the graphical execution plan.

2. Type and execute this statement to query the **charge** table:

```
SELECT member_no, SUM(charge_amt) AS Charge_SUM  
FROM dbo.charge GROUP BY member_no
```

3. Switch to the **Execution Plan** tab and view the graphical execution plan.

Did the query optimizer select the **charge** table as the source of the result set? Why or why not?

No. The charges for each member were summed by member_no when the indexed view was created. Instead of recomputing them, it is more efficient to look up those values in the nc_mem_chg_amt index.

Trainer Materials
for Microsoft Certified
Trainer Use Only

Exercise 2

Comparing I/O for Queries That Are Covered or Not Covered by Indexes

In this exercise, you will compare the I/O required when clustered and nonclustered indexes are used to retrieve selective data.

You can open, review, and execute sections of the `Covered_Queries.sql` script file in `C:\Moc\SQL2073A\Labfiles\L13`, or type and execute the provided Transact-SQL statements.

► To create a clustered index

In this procedure, you will drop all existing indexes on the **charge** table and create a clustered index on the **member_no** column of the **charge** table in the **credit** database.

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

Option	Value
User name	SQLAdminx (where <i>x</i> corresponds to your computer name as designated in the nwtraders.msft classroom domain)
Password	password

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

You have permission to log in to and administer SQL Server because you are logged as **SQLAdminx**, 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 drop existing indexes on the **charge** table:

```
USE credit
EXEC index_cleanup charge
```

4. Type and execute this statement to create a clustered index on the **member_no** column of the **charge** table:

```
CREATE CLUSTERED INDEX charge_member_no_CL
ON charge(member_no)
```

► **To evaluate the difference in execution plans when a query is covered or not covered by an index**

In this procedure, you will execute a query that returns all columns, and you will view the execution plan. Then, you will drop existing clustered indexes and create a nonclustered index, re-execute the query, and evaluate the difference in the execution plan.

1. In SQL Query Analyzer, on the **Query** menu, click **Show Execution Plan**.
2. Type and execute this statement to set the statistics option ON:

```
SET STATISTICS IO ON
```

3. Type and execute this SELECT statement to retrieve all columns for member number 5001:

```
SELECT * FROM charge WHERE member_no = 5001
```

4. Record the statistical information in the following table.

Information	Result
Scan count	1
Logical reads	3
Execution plan (index or table scan)	Index
Execution plan (type of index operation)	Clustered Index Seek

5. Type and execute these statements to drop the clustered index and create a nonclustered index on the **member_no** column of the **charge** table:

```
EXEC index_cleanup charge
CREATE NONCLUSTERED INDEX charge_member_no
ON charge(member_no)
```

6. Re-execute this SELECT statement to retrieve all columns for member number 5001:

```
SELECT * FROM charge WHERE member_no = 5001
```

7. Record the statistical information in the following table.

Information	Result
Scan count	1
Logical reads	13
Execution plan (index or table scan)	Index
Execution plan (type of index operation)	Index Seek

Both queries use an index to locate the records. Why did the nonclustered index require more logical reads?

After finding the member in the nonclustered index, SQL Server retrieves the entire data for each row from the clustered index. This action requires reading the nonclustered and clustered indexes.

► **To repeat the test with a query that is covered by the nonclustered index**

In this procedure, you will drop existing indexes, create a clustered index on the **member_no** column of the **charge** table, execute a query that is covered by the clustered index, and view the execution plan. Then, you will drop the clustered index, create a nonclustered index on the **member_no** column of the **charge** table, re-execute the query (which is also covered by the nonclustered index), and evaluate the difference in the execution plan.

1. Type and execute these statements to drop existing indexes and create a clustered index on the **member_no** column of the **charge** table:

```
EXEC index_cleanup charge
```

```
CREATE CLUSTERED INDEX charge_member_no_CL
ON charge(member_no)
```

2. Type and execute this SELECT statement to retrieve only the **member_no** column for member number 5001:

```
SELECT member_no FROM charge WHERE member_no = 5001
```

3. Record the statistical information in the following table.

Information	Result
Scan count	1
Logical reads	3
Execution plan (index or table scan)	Index
Execution plan (type of index operation)	Clustered Index Seek

4. Type and execute these statements to drop the clustered index and create a nonclustered index on the **member_no** column of the **charge** table:

```
EXEC index_cleanup charge
```

```
CREATE NONCLUSTERED INDEX charge_member_no
ON charge(member_no)
```

5. Re-execute this SELECT statement to retrieve only the **member_no** column for member number 5001:

```
SELECT member_no FROM charge WHERE member_no = 5001
```

Information	Result
Scan count	1
Logical reads	2
Execution plan (index or table scan)	Index
Execution plan (type of index operation)	Index Seek

Did the amount of I/O differ between the two queries executed by using the clustered index, even though one of the queries was covered by the clustered index? Why?

No. Because of the characteristics of a clustered index, the leaf level is the same as the data pages. A query covered by the clustered index is not beneficial.

Did the amount of I/O differ between the two queries executed by using the nonclustered index, even though one of the queries was covered by the nonclustered index? Why?

Yes. Because of the characteristics of a nonclustered index, the leaf level is different from the data pages. A query covered by the nonclustered index is beneficial because the data pages never have to be accessed. Because the query was covered by the nonclustered index, I/O was reduced from 13 to 2.

Is the performance of the query covered by the clustered index significantly different from the query covered by the nonclustered index?

No. In this case, the queries are almost identical, except for one I/O, which is not a significant performance gain.

Trainer Material
for Microsoft Certified
Trainer Use Only

Exercise 3

Comparing I/O for Queries That Retrieve a Range of Data

In this exercise, you will compare the I/O required for when clustered and nonclustered indexes are used to retrieve a range of data.

You can open, review, and execute sections of the `Range_Queries.sql` script file in `C:\Moc\SQL2073A\Labfiles\L13`, or type and execute the provided Transact-SQL statements.

► **To compare the use of a clustered index and a nonclustered index that covers a query**

In this procedure, you will drop all existing indexes, create a clustered index on the **member_no** column of the **charge** table, execute a query, and view the execution plan. You then will drop existing clustered indexes and create a nonclustered index, re-execute the query, and evaluate the difference in the execution plan.

1. Type and execute these statements to drop existing indexes and create a clustered index on the **member_no** column of the **charge** table:

```
USE credit  
EXEC index_cleanup charge
```

```
CREATE CLUSTERED INDEX charge_member_no_CL  
ON charge(member_no)
```

2. In the query window, on the **Query** menu, click **Show Execution Plan**.
3. Type and execute this statement to set the statistics option ON:

```
SET STATISTICS IO ON
```
4. Type and execute this SELECT statement to retrieve member numbers 5001 to 6000:

```
SELECT member_no FROM charge WHERE member_no  
BETWEEN 5001 AND 6000
```

5. Record the statistical information in the following table.

Information	Result
Scan count	1
Logical reads	55
Execution plan (index or table scan)	Index
Execution plan (type of index operation)	Clustered Index Seek

6. Type and execute these statements to drop the clustered index and create a nonclustered index on the **member_no** column of the **charge** table:

```
EXEC index_cleanup charge
```

```
CREATE NONCLUSTERED INDEX charge_member_no  
ON charge(member_no)
```

7. Re-execute this SELECT statement to retrieve member numbers 5001 to 6000:

```
SELECT member_no FROM charge WHERE member_no  
BETWEEN 5001 AND 6000
```

8. Record the statistical information in the following table.

Information	Result
Scan count	1
Logical reads	16
Execution plan (index or table scan)	Index
Execution plan (type of index operation)	Index Seek

9. Compare the statistics output from both queries.

Is the performance of one index significantly greater than the other index in this example? Why?

Yes. The difference in I/O will be proportional to the difference between the number of rows per data page and the number of rows per leaf-level page. For example, if you can fit 10 rows per data page and 100 rows per leaf-level page, your ratio will be 10 to 1. That means that for every 10 I/O needed to access the data pages, you would only need one I/O to access the leaf level of a covered query.

► **To execute a covered query that does not contain a WHERE clause**

In this procedure, you will drop all existing indexes, create a clustered index on the **member_no** column of the **charge** table, execute a query, and view the execution plan. Then, you will drop existing clustered indexes and create a nonclustered index, re-execute the query, and evaluate the difference in the execution plan.

1. Type and execute these statements to drop existing indexes and create a clustered index on the **member_no** column of the **charge** table:

```
EXEC index_cleanup charge
```

```
CREATE CLUSTERED INDEX charge_member_no_CL  
ON charge(member_no)
```

2. Type and execute this SELECT statement to retrieve all member numbers:

```
SELECT member_no FROM charge
```

3. Record the statistical information in the following table.

Information	Result
Scan count	1
Logical reads	675
Execution plan (index or table scan)	Index
Execution plan (type of index operation)	Clustered Index Scan

4. Type and execute these statements to drop the clustered index and create a nonclustered index on the **member_no** column of the **charge** table:

```
EXEC index_cleanup charge
```

```
CREATE NONCLUSTERED INDEX charge_member_no  
ON charge(member_no)
```

5. Re-execute this SELECT statement to retrieve all member numbers:

```
SELECT member_no FROM charge
```

6. Record the statistical information in the following table.

Information	Result
Scan count	1
Logical reads	187
Execution plan (index or table scan)	Index
Execution plan (type of index operation)	Index Scan

7. Compare the statistics output from both queries.

What is the difference between a table scan and an index scan?

A table scan always scans through the entire table. An index scan (clustered index or nonclustered index that covers a query) scans the entire leaf-level pages or scans only part of the leaf-level pages. The query optimizer always performs an index scan rather than a table scan because an index scan limits the number of pages read in most cases.

When looking at the statistics output, what is the size (number of pages) of the leaf level of the nonclustered index?

The size of the leaf level of the nonclustered index is approximately 186 pages (187 pages – 1 root page = 186).

If Time Permits:

Using Optimizer Hints to Force the Use of an Index or Join

In this exercise, you will use optimizer hints to force the query optimizer to use indexes and joins that you specify.

You can open, review, and execute sections of the Hints.sql script file in C:\Moc\SQL2073A\Labfiles\L13, or type and execute the provided Transact-SQL statements.

► To compare execution plans using an index hint

In this procedure, you will force the query optimizer to use a specific index.

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

Option	Value
User name	SQLAdminx (where <i>x</i> corresponds to your computer name as designated in the nwtraders.msft classroom domain)
Password	password

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

You have permission to log in to and administer SQL Server because you are logged as **SQLAdminx**, 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 these statements to drop existing indexes and create a clustered and nonclustered index on the **charge** table in the **credit** database:

```
USE credit
GO
```

```
EXEC index_cleanup charge
```

4. Type and execute this statement to create a clustered index on the **charge_dt** column of the **charge** table:

```
CREATE CLUSTERED INDEX charge_date_CL
ON charge(charge_dt)
```

5. Type and execute this statement to create a nonclustered index on the **member_no** column of the **charge** table:

```
CREATE NONCLUSTERED INDEX charge_member_NC
ON charge(member_no)
```

6. On the **Query** menu, select **Show Execution Plan** to turn on the graphical execution plan.

7. Type and execute this SELECT statement to retrieve rows from the **charge** table where the **member_no** equals 4000:

```
SELECT * FROM charge
WHERE member_no = 4000
```

When looking at the output from the execution plan, what index does the query optimizer use?

The charge_member_NC index is selected because it is highly selective for this query.

8. Type and execute this SELECT statement to retrieve rows from the **charge** table where the **member_no** equals 4000 while using the **charge_date_CL** index.

```
SELECT * FROM charge WITH(INDEX (charge_date_CL))
WHERE member_no = 4000
```

When looking at the output from the execution plan, what index does the query optimizer use?

The charge_date_CL index is selected because it is forced.

► To compare the execution plan by using an index hint

In this procedure, you will see how an index hint forces the query optimizer to provide a different execution plan.

1. Type and execute these statements to drop existing indexes and create a clustered index on the **charge** table in the **credit** database.

```
USE credit
GO
```

```
EXEC index_cleanup member
CREATE UNIQUE CLUSTERED INDEX member_no_CL
ON member(member_no)
```

2. In SQL Query Analyzer, turn on the graphical execution plan, and then, on the **Query** menu, select **Show Execution Plan**.

3. Type and execute this SELECT statement to join the **charge** and **member** tables

```
SELECT m.lastname, SUM(charge_amt)
FROM charge AS c JOIN member AS m
    ON c.member_no = m.member_no
WHERE m.lastname = 'BARR'
GROUP BY m.lastname
```

When looking at the execution plan output, what join method does the query optimizer use?

A merge join/inner join is selected as the most efficient.

4. Type and execute this SELECT statement to force the query optimizer to use the hash join method to join the **charge** and **member** tables:

```
SELECT m.lastname, SUM(charge_amt)
FROM charge AS c INNER HASH JOIN member AS m
    ON c.member_no = m.member_no
WHERE m.lastname = 'BARR'
GROUP BY m.lastname
```

When looking at the execution plan output, what join method does the query optimizer use?

A merge join/inner join is selected as the most efficient.

Trainer Materials
for Microsoft Certified
Trainer Use Only

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 the Query Optimizer
- Obtaining Query Plan Information
- Using an Index to Cover a Query
- Indexing Strategies
- Overriding the Query Optimizer

Delivery Tip

Use these questions to review module topics.

Ask students whether they have any questions.

1. A financial analyst performs long-running queries that slow down response time for the transaction entry staff. You ask the financial analyst to limit activity, but the financial analyst cannot tell which queries use more resources than others. What can you do to reduce the effect on the transaction query staff?

Use the query governor to limit financial analysis queries to 30 seconds.

2. With SQL Profiler, you identify the five worst performing queries. How can you determine the cause of poor query performance? What benefit does each method provide?

Graphical execution plans are one of the primary tools that indicate the cause of poor query performance. Graphical execution plans allow you to view how the query was executed. You can view each step of the plan and the sequence in which it executed. You also can view cost estimates and warnings if indexes or statistics are missing. You can also use STATISTICS IO and STATISTICS TIME to view additional information about the query, including the number of times that the table was scanned and the total I/O that SQL Server used to process the query. STATISTICS TIME lets you view how much time it takes to process each stage for the query, including CPU and compile time.

3. You have determined that by adding one more index to a table, the index can cover several queries. Having an index that covers a query increases your performance and outweighs the cost of having the additional index. To cover an index, what requirements must you meet?

All columns referenced in the query must be indexed. At least one nonclustered index must exist. In addition, covering can use composite indexes and clustered indexes.

4. While examining your indexes, you notice that the clustered index of your **Client** table is on the **Last Name** column. You know that you typically look up clients individually by last names. You also know that you frequently group clients by the **Client Representative ID** column for reporting. Should you create a nonclustered index on the **Client Representative ID** column?

No. You first should drop the clustered index on Last Name and create a new clustered index on the Client Representative ID. Creating a new clustered index greatly improves your reporting. You then should create a nonclustered index on the Last Name column. Creating a nonclustered index gives the same performance when looking up single clients by Last Name.

5. In July, you used optimizer hints in a query to improve performance. Three months later, you discover that this query again performs poorly. What is the cause?

Data can change in such a way that the optimizer hint that you specified no longer processes the query efficiently. The parameter value characteristics that users pass to the query also can change, causing the optimizer hint to perform inefficiently.

Train for Microsoft Certified
Trainer Only