MICROSOFT
TRAINING
AND CERTIFICATION

Microsoft Official Curriculum

Module 14: Analyzing Queries

Contents

Microsoft

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

Instructor Notes

Presentation:
60 Minutes

Lab:
90 Minutes

This module provides students with in-depth knowledge of how the query optimizer in Microsoft® SQL Server™ 2000 evaluates and processes queries that contain the AND and OR operators and join operations, and whether students should override the query optimizer.

In the labs, students will execute various queries and analyze how the query optimizer processes queries that contain the AND and OR logical operators. Students will also analyze how nested loop, merge, and hash joins are processed.

After completing this module, the students will be able to:

- Analyze the performance gain of writing efficient queries while creating useful indexes for queries that contain the AND logical operator.
- Analyze the performance gain of writing efficient queries while creating useful indexes for queries that contain the OR logical operator.
- Evaluate how the query optimizer uses different join strategies for query optimization.

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_14.ppt
- The C:\Moc\2073A\Demo\D14_Ex.sql example file, which contains all of the example scripts from the module, unless otherwise noted in the module.

Preparation Tasks

To prepare for this module, you should:

- Read all of the materials for this module.
- Complete the labs.
- Practice the presentation, including the animated slides.
- Review any relevant white papers located on the Trainer Materials compact disc.

Multimedia Presentation

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

You must play the multimedia presentation in class because the content is not presented anywhere else in the module. This multimedia presentation introduces how the query optimizer processes a merge join.

How Merge Joins Are Processed

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

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

Script for Multimedia Presentation

When the query optimizer processes a query that contains a join operation, it may choose to optimize the query by using a merge join algorithm.

1. In this example, the **member** table and **payment** table are joined on the **member_no** columns. The columns used to join the tables are converted to two lists of ordered input values by the merge join operation.

   Notice that the input values in **m.member_no** are unique values, whereas the input values in **p.member_no** are not unique. These tables represent a one-to-many relationship: For one member, there are many payments.

2. The query optimizer compares the first value from Input A to the first value in Input B.

   - If the values are equal, the matching rows will be returned.
   - If the value from Input A is less than the value from Input B, then the query optimizer gets the next input value from Input A.
   - If the value from Input B is less than the value from Input A, then the query optimizer gets the next input value from Input B.

   Because 1 is the lower value, 1 is discarded and the next value in Input A is used for the next comparison.

3. The query optimizer evaluates that 2 is equal to 2. When input values are equal, matching rows are returned.

4. The next input value (3) in Input A is compared with the input value 2 in Input B. Because 2 is the lower value, 2 is discarded and the next value in Input B is used for the next comparison.
5. Again, because 2 is the lower value, 2 is discarded and the next value in Input B is used for the next comparison.
6. The query optimizer continues with the evaluation process. It compares the next input value in Input B, which is also 2. Input value 2 is discarded because it is the lower value, and the next input value in Input B is used for the next comparison.
7. The next input value in Input B is 4. The query optimizer compares the two values. Because 3 are fewer than 4, 3 is discarded and the next value in Input A is used for the next comparison.
8. The query optimizer evaluates that 4 is equal to 4 and returns all matching rows.
9. The next input value (5) in Input A is compared with input value 4 in Input B. Because 4 is the lower value, 4 is discarded and the next value in Input B is used for the next comparison.
10. Again, because 4 is the lower value, 4 is discarded and the next value in Input B is used for the next comparison.
11. The next input value in Input B is 8. The query optimizer compares the two values. Because 5 are fewer than 8, 5 is discarded and the next value in Input A is used for the next comparison.
12. The query optimizer continues with the evaluation process. It compares the next input value in Input A, which is 6. It discards that input value, and uses the next input value in Input A for the next comparison.
13. As you can see, the query optimizer continues down Input A, comparing the next key value to input value 8 and discarding the lower value. The query optimizer continues with this process until it finds input values that match.
14. Because 8 is equal to 8, the query optimizer returns the rows.
15. The next input value in Input A is 9. Because 8 is the lower value, the query optimizer discards that input value, and uses the next input value in Input B for the next comparison.
16. The query optimizer compares the input value in Input B, which is also 8. It discards that input value.
17. Because all rows from Input B have been processed, the query optimizer stops evaluating the remaining values in Input A.
18. The results from the merge join operation can then be used by the next step in the execution plan, such as a Bookmark Lookup or another join operation.

Module Strategy

Use the following strategy to present this module:

- Queries That Use the AND Operator

  Describe how the query optimizer processes the AND operator. Explain that query optimization depends on whether indexes exist on some or all columns referenced in the WHERE clause.
- Queries That Use the OR Operator

  Describe how the query optimizer processes the OR operator. Explain that query optimization depends on whether indexes exist on some or all columns referenced in the WHERE clause.
- Queries That Use Join Operations

  Explain the selectivity and density of a JOIN clause. Describe how the query optimizer uses different join strategies to process queries that use joins.

Customization Information

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

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

Lab Setup

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

Setup Requirement

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

- Complete the prior lab
- Execute the C:\Moc\2073A\Batches\Restore14A.cmd batch file.

Setup Requirement

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

- Complete the prior lab
- Execute the C:\Moc\2073A\Batches\Restore14B.cmd batch file.

Warning If this course has been customized, students must execute the C:\Moc\2073A\Batches\Restore14A.cmd batch file to ensure that the first lab will function properly.

If this course has been customized, students must execute the C:\Moc\2073A\Batches\Restore14B.cmd batch file to ensure that the second lab will function properly.

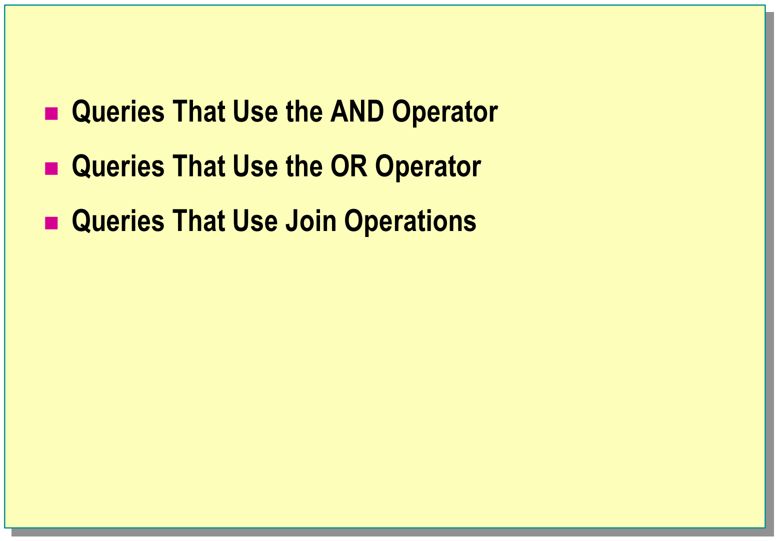
Lab Results

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

Overview

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

Lead-in
In this module, you will learn about how the query optimizer processes the AND and OR logical operators, and how the query optimizer uses different join strategies.



- **Queries That Use the AND Operator**
- **Queries That Use the OR Operator**
- **Queries That Use Join Operations**

After completing this module, you will be able to:

- Analyze the performance gain of writing efficient queries and creating useful indexes for queries that contain the AND logical operator.
- Analyze the performance gain of writing efficient queries and creating useful indexes for queries that contain the OR logical operator.
- Evaluate how the query optimizer uses different join strategies for query optimization.

Queries That Use the AND Operator

Topic Objective
To describe how a query containing the AND operator is optimized.

Lead-in
How the query optimizer processes the AND operator depends on whether indexes exist on some or all columns referenced in the WHERE clause.

- **Processing the AND Operator**
  - Returns rows that meet all conditions for every criterion specified in the WHERE clause
  - Progressively limits the number of rows returned with each additional search condition
  - Can use an index for each search condition of the WHERE clause
- **Indexing Guidelines and Performance Considerations**
  - Define an index on one highly selective search criterion
  - Evaluate performance between creating multiple, single-column indexes and creating a composite index

How query optimizer processes the AND operator depends on whether indexes exist on some, or all columns referenced in the WHERE clause.

Processing the AND Operator

When a query contains the AND operator, the query optimizer:

- Returns rows that meet all conditions for every criterion specified in the WHERE clause.
- Progressively limits the number of rows returned with each additional search condition.
- Can use an index for each search condition of the WHERE clause.
- Always uses an index if the index is useful.

  If indexes are not useful for any of the columns in the WHERE clause, the query optimizer performs a table scan or clustered index scan.
- May use multiple indexes if they are useful.

  If multiple indexes exist and some indexes are useful for any of the columns in the WHERE clause, the query optimizer determines which combination of indexes to use.

  The execution plan may show that one or most of the indexes were used to process the query. The combination of indexes is determined by the:

  - Selectivity of the search.
  - Type of indexes that exist, such as clustered or nonclustered.
  - Ability to cover the index.
  - Existence of an indexed view.

- May use only one index even though multiple useful indexes exist.

  If the query optimizer finds one index that is highly selective, it uses that index. Then, it uses the filter operation to process the remaining search conditions against the qualifying rows.

Indexing Guidelines and Performance Considerations

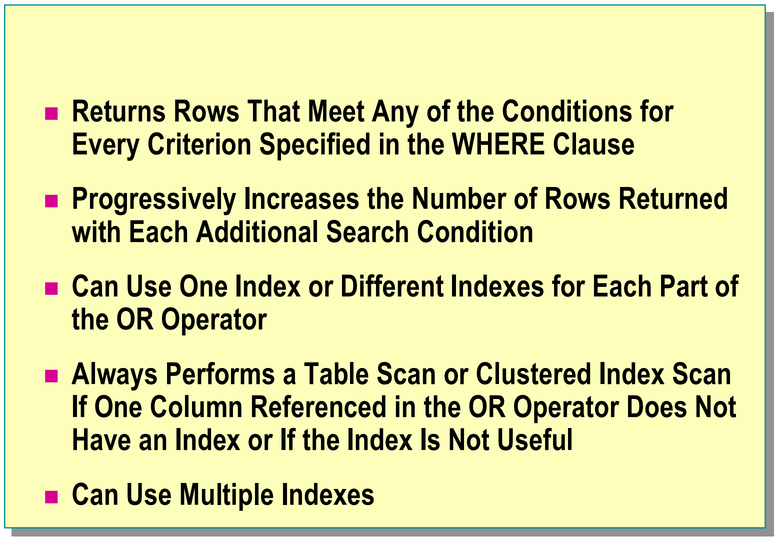
The best way to index for queries that contain the AND operator is to have at least one highly selective search criterion and define an index on that column.

You may want to compare the difference in performance when creating multiple, single-column indexes and a composite index. You do not necessarily improve query performance by indexing every column that is part of the AND operator. However, you can benefit from having multiple indexes if the columns referenced by the AND operator are of lower selectivity.

Queries That Use the OR Operator

Topic Objective
To describe how a query containing the OR operator is optimized.

Lead-in
How the query optimizer processes the OR operator also depends on whether indexes exist on some or all columns referenced in the WHERE clause.



- **Returns Rows That Meet Any of the Conditions for Every Criterion Specified in the WHERE Clause**
- **Progressively Increases the Number of Rows Returned with Each Additional Search Condition**
- **Can Use One Index or Different Indexes for Each Part of the OR Operator**
- **Always Performs a Table Scan or Clustered Index Scan If One Column Referenced in the OR Operator Does Not Have an Index or If the Index Is Not Useful**
- **Can Use Multiple Indexes**

How the query optimizer processes the OR operator also depends on whether indexes exist on some or all columns referenced in the WHERE clause.

When a query contains the OR operator, the query optimizer:

- Returns rows that meet any of the conditions for every criterion specified in the WHERE clause.
- Progressively increases the number of rows returned with each additional search condition.
- Can use one index that satisfies all parts of the OR operator, or uses different indexes for each part of the OR operator.
- Always performs a table scan or clustered index scan if one column referenced in the OR operator does not have an index, or if the index is not useful.
- If multiple indexes exist and all indexes are useful, the query optimizer:
  - Searches a table by using an index for each column.
  - Sorts the qualifying values for each column.
  - Combines the results.
  - Retrieves the qualifying rows by using the Bookmark Lookup operation.

Note The query optimizer converts the IN clause to the OR operator.

Lab A: Analyzing Queries That Use the AND and OR Operators

Topic Objective
To introduce the lab.

Lead-in
In this lab, you will observe the execution plan and evaluate how the query optimizer optimized a query that uses the AND and OR operators.

Explain the lab objectives.

Objectives

After completing this lab, you will be able to:

- Interpret statistical information on a query that uses the AND operator and determine why the query optimizer did or did not use specific indexes.
- Interpret why the query optimizer processes queries that contain a small list of values differently from queries that contain a large list of values.
- Account for the amount of input/output (I/O) used to process a query that contains nested SELECT statements, and explain why the query optimizer selected a specific execution plan.

Prerequisites

Before working on this lab, you must have:

- The **credit** database in Microsoft® SQL Server™ 2000.
- Script files, for this lab, which are located in C:\Moc\2073A\Labfiles\L14.

Lab Setup

To complete this lab, you must have either:

- Completed the prior lab, or
- Executed the C:\Moc\2073A\Batches\Restore14A.cmd batch file.

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

For More Information

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

Other resources that you can use include:

- The **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: 60 minutes

Exercise 1
Evaluating Queries That Use Some Indexes

In this exercise, you will create three indexes on the **member** table. You will execute a query that contains three search conditions by using the AND operator and thereby explain why the query optimizer created the type of plan that it did. You will also account for the I/O used to process the query.

You can open, review, and execute sections of the EvalQuery.sql script file in C:\Moc\2073A\Labfiles\L14, or type and execute the provided Transact-SQL statements.

► To create indexes

In this procedure, you will drop any indexes on the **member** table and create three nonclustered indexes on the **firstname**, **corp_no**, and **member_no** columns.

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

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

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

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

3. With SQL Query Analyzer, type and execute this statement to drop existing indexes on the **member** table:

```
USE credit
EXEC index_cleanup member
```

4. Type and execute this statement to create three indexes on the **member** table:

```
USE Credit
CREATE NONCLUSTERED INDEX fname   ON member(firstname)
CREATE NONCLUSTERED INDEX corp_no ON member(corp_no)
CREATE NONCLUSTERED INDEX mem_no  ON member(member_no)
GO
```

Note The **member** table contains 10,000 unique members (rows).

► To execute a query that uses some indexes

In this procedure, you will set the statistics option to ON, execute a query that contains three search conditions in the WHERE clause (where each column referenced in the WHERE clause has an index created for it), record the statistical information, and observe the results of the execution plan.

1. Type and execute this statement to set the statistics option to ON:

   ```
   SET STATISTICS IO ON
   ```

2. In the Query window, on the **Query** menu, click **Show Execution Plan**.
3. Type and execute this SELECT statement to retrieve data for members whose first names begin with the letter Q, whose corporate numbers are greater than 450, and whose member numbers are greater than 6000:

   ```
   USE credit
   SELECT * FROM member WHERE firstname LIKE 'Q%'
             AND corp_no > 450
             AND member_no > 6000
   ```

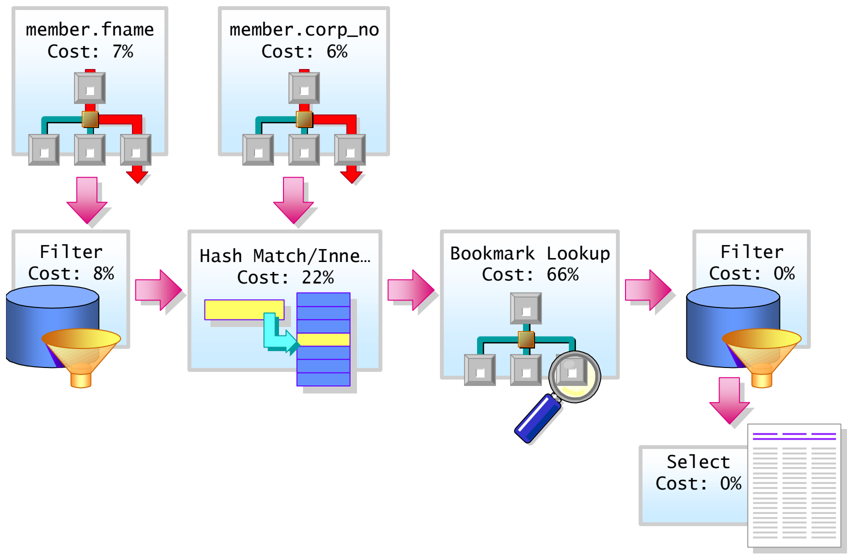
Note This query is referred to as the *original query* throughout this exercise.

4. Record the statistical information in the following table.

Information	Result
Number of rows affected	**5**
Scan count	**2**
Number of logical reads	**15**
Number and name of indexes used to process the query	**2 (fname and corp_no)**

5. Click the **Execution Plan** tab to display the execution plan graphically.
6. Examine the execution plan. It will look similar to the illustration that follows.

   Notice the operations used in the execution plan. You will analyze this execution plan throughout this exercise to account for the number of I/O used and the reasons why this execution plan was selected.



► To account for the I/O used to process the original query

In this procedure, you will rewrite the original query as three SELECT statements to understand and account for the 15 I/O of the original query. Each statement represents one search condition of the WHERE clause that limits the search. For each statement, you will record statistical information and account for the number of I/O used to process the query.

1. Type and execute this SELECT statement, which includes only the first search condition of the WHERE clause in the original query:

```
USE credit
SELECT firstname FROM member WHERE firstname LIKE 'Q%'
```

2. Record the statistical information in the following table.

Information	**Result**
Number of rows affected	**375**
Scan count	**1**
Number of logical reads	**3**
Number and name of indexes used to process the query	**1 (fname)**
Is the query covered by an index?	**Yes**

3. Type and execute this SELECT statement, which includes only the second search condition of the WHERE clause in the original query:

```
USE credit
SELECT corp_no FROM member WHERE corp_no > 450
```

4. Record the statistical information in the following table.

Information	Result
Number of rows affected	**316**
Scan count	**1**
Number of logical reads	**3**
Number and name of indexes used to process the query	**1 (corp_no)**
Is the query covered by an index?	**Yes**

5. Type and execute this SELECT statement, which includes only the third search condition of the WHERE clause in the original query:

```
USE credit
SELECT member_no FROM member WHERE member_no > 6000
```

6. Record the statistical information in the following table.

Information	Result
Number of rows affected	**4000**
Scan count	**1**
Number of logical reads	**9**
Number and name of indexes used to process the query	**1 (mem_no)**
Is the query covered by an index?	**Yes**

7. Compare the statistical information for the original query with the breakdown of each search condition of that query.

Original Query

```
USE credit
SELECT *FROM member WHERE firstname LIKE 'Q%'
          AND corp_no > 450
          AND member_no > 6000
```

Information	Result of original query	Result of query containing first search condition	Result of query containing second search condition	Result of query containing third search condition
Number of rows affected	5	375	316	4000
Scan count	2	1	1	1
Number of logical reads	15	3	3	9
Number and name of indexes used to process the query	2 (**fname** and **corp_no**)	1 (**fname**)	1 (**corp_no**)	1 (**mem_no**)

Note Your statistical information may vary from that presented in the table.

Why did each of the three individual queries use so little I/O?

The I/O was reduced because each query was able to cover the index. Reading only the leaf-level pages of the index, and not the data pages, will reduce I/O.

____

____

____

In the original query, why did the query optimizer not use the index on the **member_no** column?

Because the WHERE member_no > 6000 search condition has low selectivity. The query returns 4,000 rows out of 10,000. This WHERE clause does not limit the search, compared with other search conditions in the WHERE clause.

____

____

____

____

► To understand the execution plan by combining search conditions

In this procedure, you will rewrite the original query by combining the search conditions of the WHERE clause that the query optimizer used in the execution plan. You will record the statistical information, and evaluate the execution plan to account for the number of I/O.

1. Type and execute this SELECT statement, which includes the first and second search conditions of the WHERE clause in the original query:

```
USE credit
SELECT firstname FROM member
WHERE firstname LIKE 'Q%' AND corp_no > 450
```

2. Record the statistical information in the following table.

Information	Result
Number of rows affected	**9**
Scan count	**2**
Number of logical reads	**6**
Number and name of indexes used to process the query	**2 (fname and corp_no)**
Is the query covered by an index?	**Yes**

Notice that when you execute the query, the query optimizer uses an index for each search condition.

3. Compare the statistical information for the original query with the query that contains the first and second search conditions.

Original Query

```
USE credit
SELECT * FROM member WHERE firstname LIKE 'Q%'
          AND corp_no > 450
          AND member_no > 6000
```

Information	Result of original query	Result of query containing first and second search conditions
Number of rows affected	5	9
Scan count	2	2
Number of logical reads	15	6
Number and name of indexes used to process the query	2 (**fname** and **corp_no**)	2 (**fname** and **corp_no**)

Note Your statistical information may vary from that presented in the table.

4. Click the **Execution Plan** tab to display the execution plan graphically.
5. Examine the execution plan.

   Why do both queries have a scan count of two?

   Both queries use two indexes. The query optimizer uses one index at a time. In its first pass, the query optimizer uses the index on the firstname column of the member table and automatically does a covered query. The query is covered because the key values (one scan count and three I/O) is the only information required.

   In its second pass (scan count 2), the query optimizer uses the index on the corp_no column of the member table and automatically covers the query. The query optimizer only requires three I/O for searching on each index. Searching on the two indexes makes a total of six I/O.

Exercise 2
Evaluating Queries That Use All Indexes

In this exercise, you will execute queries containing the AND operator against the **member** table and record statistical information. The query used in this exercise is identical to the query used in Exercise 1, except that one search condition has changed from **member_no** > 6500 to **member_no** > 9500. The same three nonclustered indexes exist on the **member** table, which are defined on the **firstname**, **corp_no**, and **member_no** columns.

You will also change the indexing strategy to illustrate how different indexes can reduce your I/O in a query.

You can open, review, and execute the EvalQueryIndex.sql script file in C:\Moc\2073A\Labfiles\L14, or type and execute your own Transact-SQL statements.

► To execute a query that uses all indexes

In this procedure, you will set the statistics option to ON, execute a query, record the statistical information, and observe the results of the execution plan.

1. Type and execute this SELECT statement to retrieve data for members whose first names begin with the letter Q, whose corporate numbers are greater than 450, and whose member numbers are greater than 9500:

```
USE credit
SELECT * FROM member WHERE firstname LIKE 'Q%'
          AND corp_no > 450
          AND member_no > 9500
```

Note This query is referred to as the *original query* throughout this exercise.

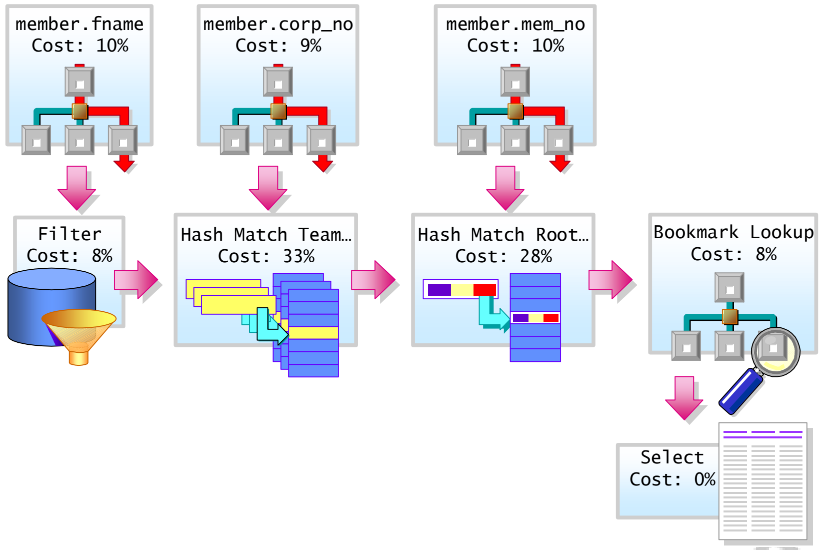
2. Record the statistical information in the following table.

Information	Result
Number of rows affected	**1**
Scan count	**3**
Number of logical reads	**10**
Number and name of indexes used to process the query	**3** (**fname**, **corp_no**, and **mem_no**)

3. Click the **Execution Plan** tab to display the execution plan graphically.

4. Examine the execution plan. It will look similar to the illustration that follows.

   Notice the operations used in the execution plan. You will be analyzing this execution plan throughout this exercise to account for the number of I/O used and the reasons why this execution plan was selected.



   Does the query optimizer use the index on the **member_no** column? Why?

   Yes. The search condition member_no > 9500 is highly selective.

5. Compare the original query in this exercise with the original query in exercise 1.

Original Query

```
USE credit
SELECT * FROM member WHERE firstname LIKE 'Q%'
          AND corp_no > 450
          AND member_no > 9500
```

Original Query (exercise 1)

```
USE credit
SELECT * FROM member WHERE firstname LIKE 'Q%'
          AND corp_no > 450
          AND member_no > 6000
```

   Why does the original query in this exercise use the index defined on the **member_no** column, whereas the original query in Exercise 1 does not?

   The search condition member_no > 6000 is of low selectivity. The query returns 4,000 rows out of 10,000 rows. The search condition member_no > 9500 is of higher selectivity and returns 500 rows out of 10,000 rows.

► **To account for the I/O used to process the original query**

In this procedure, you will rewrite the original query as three SELECT statements. Each statement will represent one search condition of the WHERE clause that limits the search. For each statement, you will record statistical information. You then will use this information to explain why the query optimizer selects a specific execution plan and to account for the number of I/O.

1. Type and execute this SELECT statement, which includes only the first search condition of the WHERE clause in the original query:

```
USE credit
SELECT firstname FROM member WHERE firstname LIKE 'Q%'
```

2. Record the statistical information in the following table.

Information	Result
Number of rows affected	**375**
Scan count	**1**
Number of logical reads	**3**
Number and name of indexes used to process the query	**1 (fname)**
Is the query covered by an index?	**Yes**

3. Type and execute this SELECT statement, which includes only the second search condition of the WHERE clause in the original query:

```
USE credit
SELECT corp_no FROM member WHERE corp_no > 450
```

4. Record the statistical information in the following table.

Information	Result
Number of rows affected	**316**
Scan count	**1**
Number of logical reads	**3**
Number and name of indexes used to process the query	**1 (corp_no)**
Is the query covered by an index?	**Yes**

5. Type and execute this SELECT statement, which includes only the third search condition of the WHERE clause in the original query:

```
USE credit
SELECT member_no FROM member WHERE member_no > 9500
```

6. Record the statistical information in the following table.

Information	Result
Number of rows affected	**500**
Scan count	**1**
Number of logical reads	**3**
Number and name of indexes used to process the query	**1 (mem_no)**
Is the query covered by an index?	**Yes**

7. Compare the statistical information for the original query with the breakdown of each search condition of that query.

Original Query

```
USE credit
SELECT * FROM member WHERE firstname LIKE 'Q%'
         AND corp_no > 450
         AND member_no > 9500
```

Information	Result of original query	Result of query containing first search condition	Result of query containing second search condition	Result of query containing third search condition
Number of rows affected	1	375	316	500
Scan count	3	1	1	1
Number of logical reads	10	3	3	3
Number and name of indexes used to process the query	3 (**fname**, **corp_no**, and **mem_no**)	1 (**fname**)	1 (**corp_no**)	1 (**mem_no**)

Note Your statistical information may vary from that presented in the table.

8. Click the **Execution Plan** tab to display the execution plan graphically.
9. Examine the execution plan for the original query.

   In the original query, how do you account for the 10 I/O?

   The query optimizer uses three indexes. Each index used requires three I/O. Three I/O multiplied by three I/O equals nine I/O. The one remaining I/O is used for the Bookmark Lookup operator, bringing the total to 10 I/O.

► To execute a query against a table with a clustered index

In this procedure, you will drop the nonclustered index on the **member.member_no** column and create a clustered index on the **member.member_no** column. After you create the index, you will execute the original query and observe the changes in the execution plan and the page I/O.

1. Type and execute this statement to drop the nonclustered index on the **member.member_no** column and create a clustered index:

```
USE credit
DROP INDEX member.mem_no
GO
CREATE CLUSTERED INDEX mem_no_CL ON member(member_no)
GO
```

Note Two nonclustered indexes exist: One is defined on the **corp_no** column, and the other is defined on the **firstname** column.

2. Type and execute the original query against the **member** table with a clustered index on the **member_no** column:

```
USE credit
SELECT * FROM member WHERE firstname LIKE 'Q%'
          AND corp_no > 450
          AND member_no > 9500
```

3. Record the statistical information in the following table.

Information	Result
Number of rows affected	**1**
Scan count	**1**
Number of logical reads	**9**
Number and name of indexes used to process the query	**1 (mem_no)**
Is the query covered by an index?	**No**

4. Compare the statistical information of the query against a table that has a clustered index on the **member_no** column with the table that has a nonclustered index on the **member_no** column.

Information	Result of query (clustered index on member_no)	Result of query (nonclustered index on member_no)
Number of rows affected	1	1
Scan count	1	3
Number of logical reads	9	10
Number and name of indexes used to process the query	1 (**mem_no**)	3 (**fname**, **corp_no**, and **mem_no**)

Note Your statistical information may vary from that presented in the table.

5. Click the **Execution Plan** tab to display the execution plan graphically.
6. Examine the execution plan.

   Has query performance improved with a clustered index on the **member_no** column? Why?

   Yes. The query with the clustered index on the member_no column only requires nine I/O, one scan count, and fewer steps in the execution plan.

   ____

   ____

   Why does the query optimizer use the clustered index on the **member_no** column?

   The query optimizer typically selects a clustered index if the query is not covered by an index. The query optimizer uses a clustered index because the rows are sorted and it is more efficient when processing queries that specify ranges of data. By using the clustered index on member_no, SQL Server reads all 500 rows, which requires 9 I/O. Rows that do not meet the search conditions are eliminated as the range of member numbers are read.

   ____

   ____

   ____

   ____

Exercise 3
Evaluating Queries That Use the IN Keyword

In this exercise, you will create an index on the **member** table, observe the performance of queries containing the IN keyword, and observe how the execution plan changes as the list of values grows in size. The larger the list, the less efficient the query becomes.

You can open, review, and execute sections of the EvalQueryIN.sql script file in C:\Moc\2073A\Labfiles\L14, or type and execute the provided Transact-SQL statements.

► To execute a query that contains an IN keyword

In this procedure, you will drop any existing indexes on the **member** table, create an index, execute a query, record the statistical information, and examine the execution plan.

1. Type and execute this statement to drop existing indexes on the **member** table:

```
USE credit
EXEC index_cleanup member
```

2. Type and execute this statement to create a unique, nonclustered index on the **member_no** column of the **member** table:

```
USE credit
CREATE UNIQUE nonclustered INDEX mbr_mem_no
   ON member(member_no)
GO
```

3. Type and execute this statement to set the statistics option to ON:

```
SET STATISTICS IO ON
```

4. Type and execute this SELECT statement to retrieve all data for specific member numbers:

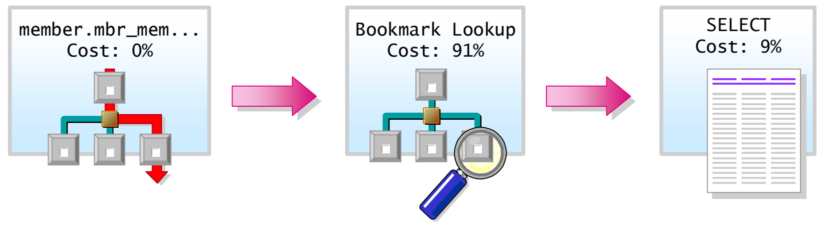
```
USE credit
SELECT * FROM member WHERE member_no
   IN (100,101,102,103,104,105,106,107,108,109,200,201,
       202,203,204,205,206,207,208,209,210,211,212,213,
       214,215,216,217,218,219,220,221,222,223,224,225,
       226,227,228,229,230,231,232)
```

5. Record the statistical information in the following table:

Information	Result
Number of rows affected	**43**
Scan count	**43**
Number of logical reads	**129**
Number and name of indexes used to process the query	**1 (mbr_mem_no)**
Is the query covered by an index?	**Yes**

6. Click the **Execution Plan** tab to graphically view the execution plan.
7. Examine the execution plan. It will look similar to the illustration that follows.

   Notice the operations used in the execution plan. You will be analyzing this execution plan throughout this exercise to account for the number of I/O used and the reasons why this execution plan was selected.



Knowing that the Bookmark Lookup accounts for 43 of the 129 I/O, how do you account for the remaining 86 I/O?

Tip STATISTICS I/O shows a scan count of 43.

The remaining 86 I/O are from reading (scan count) the table once for each value in the list of values of the query. The list of values in the query contains 43 values. The query optimizer uses the nonclustered index to search for each value. To get the necessary value, it reads the root page and then one leaf-level page to get the row identifier (RID) (covers the index). Forty-three rows, requiring two I/O each, equal 86 I/O. Using a RID to retrieve rows requires exactly one I/O for each row. Twelve rows equal 43 I/O; 43 + 86 = 129 I/O.

If the query were modified to use an index that covers the query, what step would be eliminated in the execution plan? Why?

The Bookmark Lookup step would be eliminated, because the query would only need to process the first step of the execution plan and select only the member_no column data.

What would be the total page I/O for this query, which is covered by an index?

43 page I/O. Eliminating the Bookmark Lookup eliminates 43 I/O. 129 - 43 = 86 I/O.

____

____

► **To show when a query containing a list of values becomes inefficient**

In this procedure, you will execute a query, record the statistical information, and compare the execution plan with that of the preceding query. The query used in this procedure is similar to the previous query, except that the IN list includes one additional value. The additional value in the query causes the query optimizer to process the query in a different way.

1. Type and execute this SELECT statement to retrieve specific member numbers. Notice that this query differs from the original query. There are now 44 values in the IN list:

```
USE credit
SELECT * FROM member WHERE member_no
IN (100,101,102,103,104,105,106,107,108,109,200,201,202,
    203,204,205,206,207,208,209,210,211,212,213,214,215,
    216,217,218,219,220,221,222,223,224,225,226,227,228,
    229,230,231,232,233)
```

2. Record the statistical information in the following table.

Information	Result
Number of rows affected	**44**
Scan count	**1**
Number of logical reads	**145**
Number and name of indexes used to process the query	**none**
Is the query covered by an index?	**No**

3. Compare the statistical information for this query with the statistical information in the previous query.

Previous Query

```
USE credit
SELECT * FROM member WHERE member_no
   IN (100,101,102,103,104,105,106,107,108,109,200,201,
       202,203,204,205,206,207,208,209,210,211,212,213,
       214,215,216,217,218,219,220,221,222,223,224,225,
       226,227,228,229,230,231,232)
```

Query Containing Additional Value

```
USE credit
SELECT * FROM member WHERE member_no
IN (100,101,102,103,104,105,106,107,108,109,200,201,202,
    203,204,205,206,207,208,209,210,211,212,213,214,215,
    216,217,218,219,220,221,222,223,224,225,226,227,228,
    229,230,231,232,233)
```

Information	Result of previous query	Result of query containing additional value
Number of rows affected	43	44
Scan count	43	1
Number of logical reads	129	145
Number and name of indexes used to process the query	Yes (**mem_no**)	None

4. Click the **Execution Plan** tab to display the execution plan graphically.
5. Examine the execution plan.

   Why did the query optimizer use a different execution plan for the query with the additional value (233) in the list of values?

   As the list of values becomes larger, the query optimizer processes the query in a different way. A new execution plan is used to process a larger set of values more efficiently.

   ____

   ____

   ____

   ____

   Describe the execution plan for the query containing the additional value (233)?

   The execution plan first creates an internal table to store the list of values, which are sorted (Constant Scan). The internal table is then joined using a Hash join with the list of key values retrieved by using a table scan. This execution plan requires more I/O to process the join between the internal table and the member table.

   ____

   ____

   ____

   ____

Exercise 4
Evaluating Queries That Contain Nested SELECT Statements

In this exercise, you will observe the performance of queries containing nested SELECT statements. You will examine the execution plan of a query that returns a list of values, account for the number of I/O, and explain why the query optimizer selected the specific execution plan.

You can open, review, and execute sections of the EvalQueryNested.sql script file in C:\Moc\2073A\Labfiles\L14, or type and execute the provided Transact-SQL statements.

► To execute a query containing a nested SELECT statement

In this procedure, you will execute a query that contains a nested SELECT statement that returns a list of values and record the statistical information.

1. Type and execute this SELECT statement to retrieve member data for members whose member numbers are between 100 and 111:

```
USE credit
SELECT * FROM member WHERE member_no
IN (SELECT member_no FROM member WHERE member_no
BETWEEN 100 AND 111)
```

Note This query is referred to as the original query throughout this exercise.

2. Record the statistical information in the following table.

Information	Result
Number of rows affected	**12**
Scan count	**13**
Number of logical reads	**44**
Number and name of indexes used to process the query	**1 (mbr_mem_no)**

► To account for the I/O used to process the original query

In this procedure, you will rewrite the original query as two SELECT statements. Each statement represents a step in the execution plan. For each statement, you will record statistical information, explain why the query optimizer selected a specific execution plan, and you then account for the number of I/O.

1. Type and execute this SELECT statement to retrieve member numbers between 100 and 111:

```
USE credit
SELECT member_no FROM member WHERE member_no
      BETWEEN 100 AND 111
```

Note In the original query, this SELECT statement is the first step of the execution plan.

2. Record the statistical information in the following table.

Information	Result
Number of rows affected	**12**
Scan count	**1**
Number of logical reads	**2**

Is an index used to process the query? Is this query covered by an index? Why?

Yes (member_no). Yes, this query is covered by an index, because all of the data can be found in the index.

3. Type and execute this SELECT statement to retrieve specific member numbers:

```
USE credit
SELECT member_no FROM member WHERE member_no
IN (100,101,102,103,104,105,106,107,108,109,110,111)
```

Note In the original query, this SELECT statement is the second step of the execution plan.

4. Record the statistical information in the following table.

Information	Result
Number of rows affected	**12**
Scan count	**12**
Number of logical reads	**24**

Is an index used to process the query? Is this query covered by an index?

Yes (member_no). Yes, this query is covered by an index, because all of the data can be found in the index.

5. Compare the statistical information for the original query with the breakdown of each step of the execution plan.

Original Query

```
USE credit
SELECT * FROM member WHERE member_no IN
   (SELECT member_no FROM member WHERE member_no
           BETWEEN 100 AND 111)
```

First Step Query

```
USE credit
SELECT member_no FROM member WHERE member_no
           BETWEEN 100 AND 111
```

Second Step Query

```
USE credit
SELECT member_no FROM member WHERE member_no IN
   (100,101,102,103,104,105,106,107,108,109,110,111)
```

Information	**Result of original query**	**Result of query (first step of execution plan)**	**Result of query (second step of execution plan)**
Number of rows affected	12	12	12
Scan count	13	1	12
Number of logical reads	44	2	24

Note Your statistical information may vary from that presented in the table.

6. Click the **Execution Plan** tab to graphically view the execution plan.

7. Execute the original query and examine the execution plan.

   In the original query, explain why the query optimizer created this plan, and specify how much I/O was used by each step in the execution plan.

   The query optimizer first processes the nested select by reading the range of values from the leaf level of the nonclustered index (an index covers this query). This requires only two I/O and returns 12 values (not the row). This generates one scan count.

   It then processes each of the 12 values one at a time. Each value requires two I/O—one to read the root page, and the other to read the leaf level page—which brings the total to 24 I/O. This step uses an index that covers the query. This generates 12 scan counts, which result in a total scan count of 13 (1+ 12 = 13).

   The join operation joins these results, producing 12 rows. The final step executes a Bookmark Lookup where each of the 12 values requires one I/O.

   The total I/O that this execution plan uses is 2 + 24 + 12 = 38. The six I/O used by the join operation bring the total to 44 I/O.

Exercise 5
Evaluating Queries That Contain the OR Operator

In this exercise, you will execute several queries containing the OR operator against the **member** table, which has a nonclustered index on the **member_no** column. You will record the statistical information, compare I/O, and examine the execution plan. You will drop existing indexes, create two indexes, re-execute a query, and compare the execution plan used for the same query executed against a table with partial indexing.

You can open, review, and execute sections of the EvalQueryOR.sql script file in C:\Moc\2073A\Labfiles\L14, or type and execute the provided Transact-SQL statements.

► **To execute a query against a table with partial indexing**

In this procedure, you will execute three queries and record and evaluate their statistical information.

1. Drop all indexes on the member table in the **credit** database.

```
USE credit
EXEC index_cleanup member
```

2. Create a unique nonclustered index on the **member_no** column of the **member** table.

```
CREATE UNIQUE nonclustered INDEX mbr_mem_no ON
member(member_no)
```

3. Set statistics I/O to ON.

```
SET STATISTICS IO ON
```

4. Type and execute this SELECT statement to retrieve a member where member number equals 1234, or region number equals 5:

Query 1

```
USE credit
SELECT * FROM member WHERE member_no=1234 OR region_no=5
```

Note A nonclustered index on the **member_no** column of the **member** table exists.

5. Record the statistical information in the following table.

Information	Result
Number of rows affected	**1,100**
Scan count	**1**
Number of logical reads	**145**
Number and name of indexes used to process the query	**None (table scan)**

6. Type and execute this SELECT statement to retrieve a member where member number equals 1,234, or corporate number equals 410:

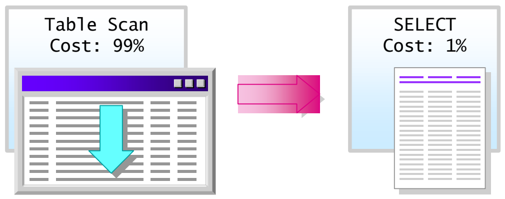
Query 2

```
USE credit
SELECT * FROM member WHERE member_no=1234 OR corp_no=410
```

7. Record the statistical information in the following table.

Information	Result
Number of rows affected	**7**
Scan count	**1**
Number of logical reads	**145**
Number and name of indexes used to process the query	**None (table scan)**

8. Click the **Execution Plan** tab to graphically view the execution plan.
9. Examine the execution plan. It will look similar to the illustration that follows.



Notice the operations used in the execution plan. You will use this execution plan to compare to another execution plan later in the exercise.

10. Type and execute this SELECT statement to retrieve a member where region number equals 5, or corporate number equals 410:

Query 3

```
USE credit
SELECT * FROM member WHERE region_no = 5 OR corp_no = 410
```

11. Record the statistical information in the following table.

Information	Result
Number of rows affected	**1105**
Scan count	**1**
Number of logical reads	**145**
Number and name of indexes used to process the query	**None (table scan)**

12. Compare the statistical information for all three queries.

Query 1

```
USE credit
SELECT * FROM member WHERE member_no=1234 OR region_no=5
```

Query 2

```
USE credit
SELECT * FROM member WHERE member_no=1234 OR corp_no=410
```

Query 3

```
USE credit
SELECT * FROM member WHERE region_no=5 OR corp_no=410
```

Information	Result of query 1	Result of query 2	Result of query 3
Number of rows affected	1,100	7	1,105
Scan count	1	1	1
Number of logical reads	145	145	145
Number and name of indexes used to process the query	None (table scan)	None (table scan)	None (table scan)

Note Your statistical information may vary from that presented in the table.

Do these three execution plans differ? Why?

No. The execution plans do not differ because they all require a table scan. At least one of the columns that the OR operator references does not have an index. If one column does not have an index, then the query optimizer performs a table scan. Because of the way that the logic of the OR operator works, all columns referencing an OR operator are processed separately.

____

____

____

____

► **To drop existing indexes and create indexes**

In this procedure, you will drop any existing indexes and create two indexes on the **member** table.

1. Type and execute this statement to set the statistics option to OFF:

```
SET STATISTICS IO OFF
```

2. Type and execute this statement to drop existing indexes on the **member** table:

```
USE credit
EXEC index_cleanup member
```

3. Type and execute this statement to create two indexes on the **member** table:

```
USE credit
CREATE UNIQUE nonclustered INDEX mbr_mem_no
   ON member(member_no)
CREATE clustered INDEX mbr_corp_no_CL
   ON member(corp_no)
```

► To execute a query against a table with complete indexing

In this procedure, you will set the statistics option to ON, re-execute query 2 of this exercise, and record and evaluate the statistical information.

1. Type and execute this statement to set the statistics option to ON:

```
SET STATISTICS IO ON
```

2. Re-execute query 2, which retrieves member number 1,234, or corporate number 410.

Query 2

```
USE credit
SELECT * FROM member WHERE member_no=1234 OR corp_no=410
```

3. Record the statistical information in the following table.

Information	Result
Number of rows affected	**7**
Scan count	**2**
Number of logical reads	**18**
Number and name of indexes used to process the query	**2 (mbr_mem_no and mbr_corp_no_CL)**

4. Compare the statistical information of query 2 against a table with partial indexing with complete indexing.

Query 2

```
USE credit
SELECT * FROM member WHERE member_no=1234 OR corp_no=410
```

Information	Result of query 2 (partial indexing)	Result of query 2 (complete indexing)
Number of rows affected	7	7
Scan count	1	2
Number of logical reads	145	18
Number and name of indexes used to process the query	None (table scan)	Yes (**mbr_mem_no** and **mbr_corp_no_CL**)

5. Click the **Execution Plan** tab to graphically view the execution plan.
6. Examine the execution plan and compare it with the execution plan for the same query executed against a table with partial indexing.

   Why is the execution plan for a table, with complete indexing different from the execution plan, used for the same query executed against a table with partial indexing?

   The execution plan for a table with complete indexing is different because an index exists and is useful for each column referenced by the OR operator. The query optimizer uses one index to search on the corp_no column and a different index to search on the member_no column.

What is the execution plan, and how many I/O does it use?

Step 1a requires two I/O to retrieve one row on the member_no column, which has a nonclustered index that covers the query. This operation does not contribute to the value of the scan count, because it covers the index. The operation does not access the table.

Step 1b requires two I/O to retrieve six rows on the corp_no column, which has a clustered index. This operation accounts for one of the scan counts.

Step 2 and 3 concatenate and sort the results. These operations require seven I/O.

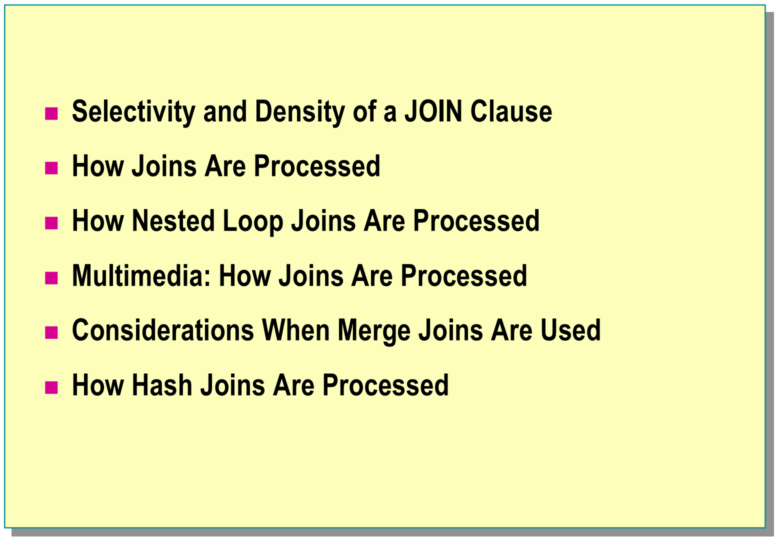
Step 3 is a Bookmark Lookup operation. The query optimizer reads each of the seven rows individually by using the RID or clustering key. One I/O for each row equals seven I/O and accounts for the second scan count.

Total I/O: 2 + 2 + 7 + 7 = 18 I/O.

◆ Queries That Use Join Operations

Topic Objective
To point out the topics in this section.

Lead-in
In this section, we will discuss query optimizations that use join operations.



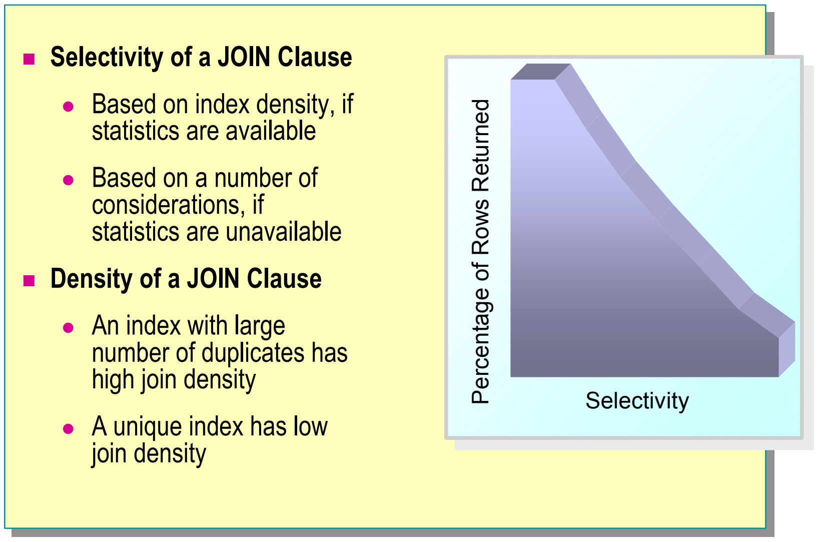
- Selectivity and Density of a JOIN Clause
- How Joins Are Processed
- How Nested Loop Joins Are Processed
- Multimedia: How Joins Are Processed
- Considerations When Merge Joins Are Used
- How Hash Joins Are Processed

In this section, we will discuss how the query optimizer optimizes queries that use join operations.

Selectivity and Density of a JOIN Clause

Topic Objective
To discuss the selectivity and density of a JOIN clause.

Lead-in
The order in which the query optimizer processes joins is determined by the existence of indexes and a WHERE clause, in addition to the selectivity and density of the data.



- **Selectivity of a JOIN Clause**
  - Based on index density, if statistics are available
  - Based on a number of considerations, if statistics are unavailable
- **Density of a JOIN Clause**
  - An index with large number of duplicates has high join density
  - A unique index has low join density

Percentage of Rows Returned
Selectivity

The order in which the query optimizer processes joins is determined by the existence of indexes and a WHERE clause, in addition to the selectivity and density of the data.

Delivery Tip
Use the slide illustration to reiterate that a low selectivity returns many rows, and a high selectivity returns few rows.

Selectivity of a JOIN Clause

The selectivity of a JOIN clause is the percentage of rows from one table that are joined to a single row from another. Selectivity is derived from the number of rows that are estimated to be returned, as seen with the WHERE clause.

A low selectivity returns many rows, and a high selectivity returns few rows. The base is the multiple of the rows in both tables after local predicates (WHERE clause) on joined tables and aggregations are applied. This algorithm is different from determining how many rows match a search condition.

How Selectivity of a JOIN Clause Is Determined

You can calculate the selectivity of a JOIN clause by using the density of the data. The query optimizer determines selectivity of a JOIN clause based on the following parameters:

- If statistics are available, join selectivity is based on the density of the index for all of the columns.
- If statistics are unavailable because indexes do not exist, existing indexes are not useful, or if a WHERE clause is not included in the query, the query optimizer processes the query more efficiently by:
  - Applying an appropriate join strategy.
  - Using other physical operators.
  - Building column statistics dynamically.
  - The number of rows in each table of the join.

Delivery Tip
Point out the considerations that the query optimizer estimates to determine the selectivity of a JOIN clause when statistics are unavailable.

Density of a JOIN Clause

The density of a JOIN clause is the average percentage of duplicates between the inner and outer tables. The query optimizer uses the density of a JOIN clause to determine which table is processed as the inner table, and which table is processed as the outer table.

- An index with a large number of duplicates has high join density, which is not very selective for joins.

  For example, the **orders_details** table contains many orders for one customer.

- A unique index has a low join density, which is highly selective.

  For example, the **customer** table lists each customer only once. The **customer ID** column is unique.

If an index has a low join density, the query optimizer can access data by using a clustered or nonclustered index. However, only a clustered index is typically useful for indexes with a high join density.

Example

In this example, use the following assumptions to determine how the query optimizer produces a execution plan:

- The **employee** table contains 1,000 rows.
- The **department** table contains 100 rows (unique departments).
- The data is evenly distributed (10 employees per department).
- No indexes or statistics exist.

```
USE credit
SELECT *
FROM department AS dept INNER JOIN employee AS empl
ON dept.deptno = empl.deptno
```

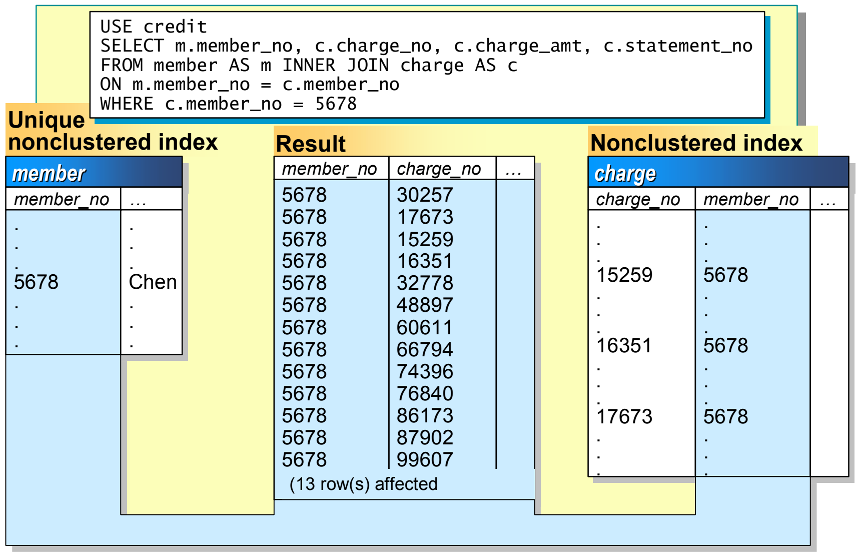
When indexes do not exist on columns that are joined, the query optimizer uses a join strategy that determines which table is the outer table and which table is the inner table. It does this by evaluating the row ratio between tables.

If any search conditions exist in the WHERE clause, the query optimizer may use these conditions first to determine how to join the tables. This determination is based on selectivity.

How Joins Are Processed

Topic Objective
To illustrate how the query optimizer processes a join.

Lead-in
Let's take a look at how the query optimizer processes a join.



```
USE credit
SELECT m.member_no, c.charge_no, c.charge_amt, c.statement_no
FROM member AS m INNER JOIN charge AS c
ON m.member_no = c.member_no
WHERE c.member_no = 5678
```

Unique nonclustered index

member

member_no	…
.	.
.	.
.	.
5678	Chen
.	.
.	.
.	.

Result

member_no	*charge_no*	…
5678	30257	
5678	17673	
5678	15259	
5678	16351	
5678	32778	
5678	48897	
5678	60611	
5678	66794	
5678	74396	
5678	76840	
5678	86173	
5678	87902	
5678	99607	

(13 row(s) affected)

Nonclustered index

charge

charge_no	*member_no*	…
.	.	
.	.	
.	.	
15259	5678	
.	.	
.	.	
.	.	
16351	5678	
.	.	
.	.	
.	.	
17673	5678	
.	.	
.	.	
.	.	

Delivery Tip
Point out that the query optimizer converts the search criteria in the WHERE clause so that the **member** table is the outer table.

Explain that by making this conversion, the query optimizer limits the search. It limits the search because the **member** table has only one qualifying row, whereas the **charge** table has many rows.

An understanding of how the query optimizer processes join operations enables you to determine what types of indexes are useful to create.

Joins are processed as pairs. Regardless of how many tables you are combining, joins are always between two tables. The result of these joins is called an *intermediate result*. Intermediate results can then be joined to another table by using any of the join algorithms. For each join, the query optimizer will determine the appropriate join algorithm to use.

When processing join operations, the query optimizer typically:

- Determines the order in which the tables are processed, based on indexes, selectivity, and density.

  Order is not determined by the order of the table referenced in the SELECT statement.
- Identifies which table is the optimal outer table.
- Finds all matching rows in inner table for each qualifying row in the outer table.

Evaluating the Use of Indexes

The selectivity and density of a JOIN clause affects which type of index is most useful for processing the query.

- An index on the column that is specified in the WHERE clause can influence which table is used as the outer table and which join strategy is used. Selectivity determines which table is the inner table.
- The query optimizer automatically considers the use of redundant JOIN clauses and conditions in the WHERE clause.

Example

In this example, there is a unique nonclustered index on the **member_no** column in the **member** table, and a nonclustered index on the **member_no** column in the **charge** table. Both indexes are useful for processing the query.

```
USE credit
SELECT m.member_no, c.charge_no, c.charge_amt, c.statement_no
FROM member AS m INNER JOIN charge AS c
ON m.member_no = c.member_no
WHERE c.member_no = 5678
```

The query optimizer converts the search criteria in the WHERE clause so that the query is processed as:

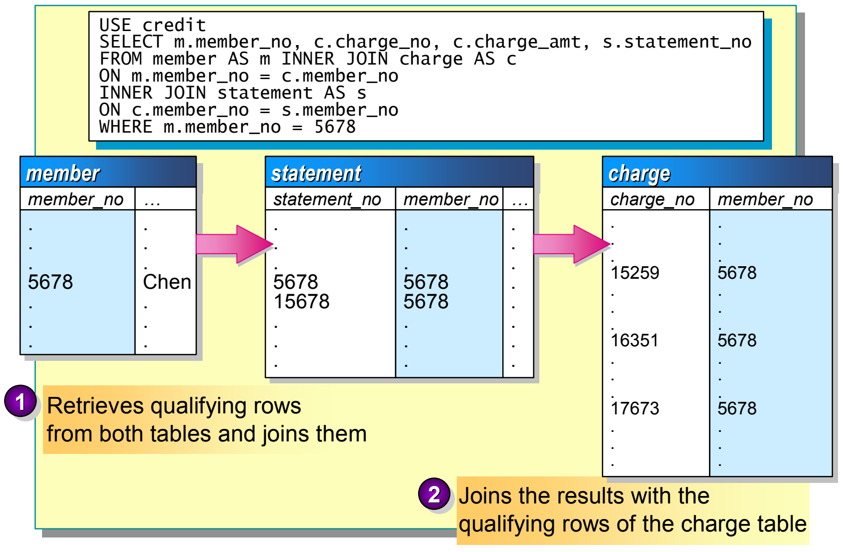
WHERE m.member_no = 5678

By converting the **member** table to the outer table, the query optimizer limits the search, because the **member** table has only one qualifying row, whereas the **charge** table has many rows.

How Nested Loop Joins Are Processed

Topic Objective
To discuss how nested joins are processed.

Lead-in
If there is a JOIN clause in the query, the query optimizer evaluates the number of tables, indexes, and joins to determine the optimal order, and what join strategy to use.



```
USE credit
SELECT m.member_no, c.charge_no, c.charge_amt, s.statement_no
FROM member AS m INNER JOIN charge AS c
ON m.member_no = c.member_no
INNER JOIN statement AS s
ON c.member_no = s.member_no
WHERE m.member_no = 5678
```

member
member_no
…
5678
Chen
statement
statement_no
member_no
…
5678
15678
5678
5678
charge
charge_no
member_no
15259
5678
16351
5678
17673
5678
1 Retrieves qualifying rows from both tables and joins them
2 Joins the results with the qualifying rows of the charge table

If there is a JOIN clause in the query, the query optimizer evaluates the number of tables, indexes, and joins to determine the optimal order, and what join strategy to use. The query optimizer processes nested loop joins as nested iterations.

Defining Nested Iteration

A *nested iteration* is when the query optimizer constructs a set of nested loops, and the result set grows as it progresses through the rows. The query optimizer performs the following steps.

1. Finds a row from the first table.
2. Uses that row to scan the next table.
3. Uses the result of the previous table to scan the next table.

Evaluating Join Combinations

The query optimizer automatically evaluates at least four or more possible join combinations, even if those combinations are not specified in the join predicate. You do not have to add redundant clauses. The query optimizer balances the cost and uses statistics to determine the number of join combinations that it evaluates. Evaluating every possible join combination is inefficient and costly.

Evaluating Cost of Query Performance

When the query optimizer performs a nested join, you should be aware that certain costs are incurred. Nested loop joins are far superior to both merge joins and hash joins when executing small transactions, such as those affecting only a small set of rows. The query optimizer:

- Uses nested loop joins if the outer input is quite small and the inner input is indexed and quite large.
- Uses the smaller input as the outer table.
- Requires that a useful index exist on the join predicate for the inner table.
- Always uses a nested loop join strategy if the join operation uses an operator other than an equality operator.

Example

In this example, the **member** table (10,000 rows) is joined to the **charge** table (100,000 rows), and the **charge** table is joined to the **statement** table (20,000 rows). Nonclustered indexes exist on the **member_no** column in each table. The query optimizer processes the join as the **member** table joined to the **statement** table, and the result of that join is combined with the **charge** table.

```
USE credit
SELECT m.member_no, c.charge_no, c.charge_amt, s.statement_no
FROM member AS m INNER JOIN charge AS c
     ON m.member_no = c.member_no
INNER JOIN statement AS s
     ON c.member_no = s.member_no
WHERE m.member_no = 5678
```

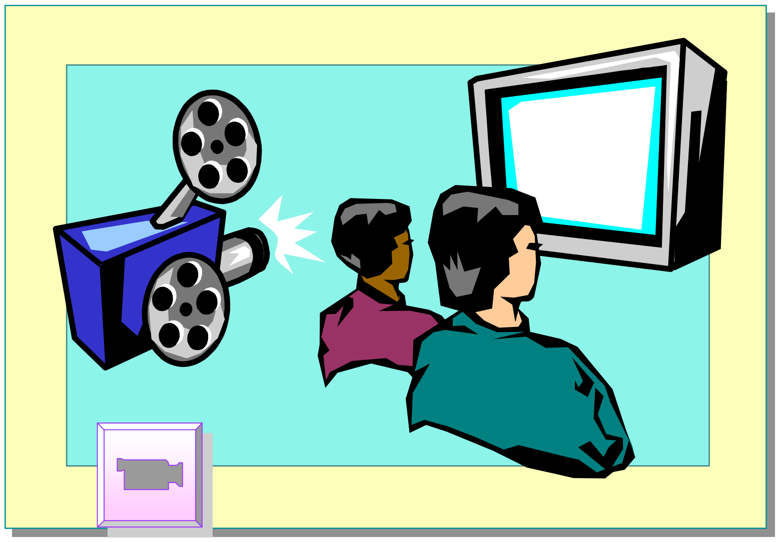
The query optimizer performs the following steps to process the query:

1. Retrieves the qualifying rows from the **member** table and the **statement** table, and then joins the result by using the nested loop join strategy.
2. Retrieves the qualifying rows from the **charge** table, and then joins that result with the results of the first nested loop join by using another nested loop join strategy.

Multimedia: How Merge Joins Are Processed

Topic Objective
To discuss how merge joins are processed.

Lead-in
In this presentation, you will see how merge joins are processed.



The columns of the join conditions are used as inputs to process a merge join. SQL Server performs the following steps when using a merge join strategy:

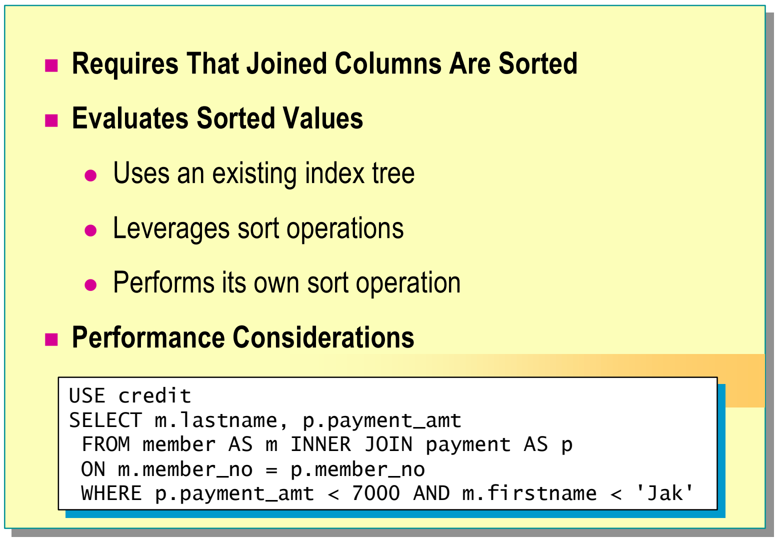
1. Gets the first input values from each input set.
2. Compares input values.
3. Performs a merge algorithm.
   - If the input values are equal, the rows are returned.
   - If the input values are not equal, the lower value is discarded, and the next input value from that input is used for the next comparison.
4. Repeats the process until all of the rows from one of the input sets have been processed.
5. Evaluates any remaining search conditions in the query and returns only rows that qualify.

Note Only one pass per input is done. The merge join operation ends after all of the input values of one input have been evaluated. The remaining values from the other input are not processed.

Considerations When Merge Joins Are Used

Topic Objective
To point out the details in the merge join operation.

Lead-in
A merge join uses two sorted inputs and then merges them.



- **Requires That Joined Columns Are Sorted**
- **Evaluates Sorted Values**
  - Uses an existing index tree
  - Leverages sort operations
  - Performs its own sort operation
- **Performance Considerations**

```
USE credit
SELECT m.lastname, p.payment_amt
 FROM member AS m INNER JOIN payment AS p
 ON m.member_no = p.member_no
 WHERE p.payment_amt < 7000 AND m.firstname < 'Jak'
```

A merge uses two sorted inputs and then merges them.

Requires That Joined Columns Are Sorted

If you execute a query with join operations, and the joined columns are in sorted order, the query optimizer processes the query by using a merge join strategy. A merge join is very efficient because the columns are already sorted, and it requires less page I/O.

Evaluates Sorted Values

For the query optimizer to use the merge join, the inputs must be sorted. The query optimizer evaluates sorted values in the following order:

1. Uses an existing index tree (most typical). The query optimizer can use the index tree from a clustered index or a covered nonclustered index.
2. Leverages sort operations that the GROUP BY, ORDER BY, and CUBE clauses use. The sorting operation only has to be performed once.
3. Performs its own sort operation in which a SORT operator is displayed when graphically viewing the execution plan. The query optimizer does this very rarely.

Example 1

In this example, a clustered index exists on the **member_no** column of the **payment** table, and a unique clustered index exists on the **member_no** column of the **member** table. The query optimizer scans the member table and the payment table by using the clustered index for each table. After scanning the contents of each table, the query optimizer performs a merge join between both tables, because both inputs are already sorted from the clustered indexes. This is a merge join/inner join.

```
USE credit
SELECT m.lastname, p.payment_amt
 FROM member AS m INNER JOIN payment AS p
      ON m.member_no = p.member_no
 WHERE p.payment_amt < 7000 AND m.firstname < 'Jak'
```

Example 2

In this example, a unique clustered index exists on the **member_no** column of the **member** table, and the query explicitly specifies an ORDER BY clause on the **member_no** column of the **payment** table.

```
USE credit
SELECT m.lastname, m.firstname, p.payment_dt
  FROM member AS m INNER JOIN payment AS p
      ON m.member_no = p.member_no
  ORDER BY p.member_no
```

Performance Considerations

Consider the following facts about the query optimizer's use of the merge join:

- SQL Server performs a merge join for all types of join operations (except cross join or full join operations), including UNION operations.
- A merge join operation may be a one-to-one, one-to-many, or many-to-many operation.

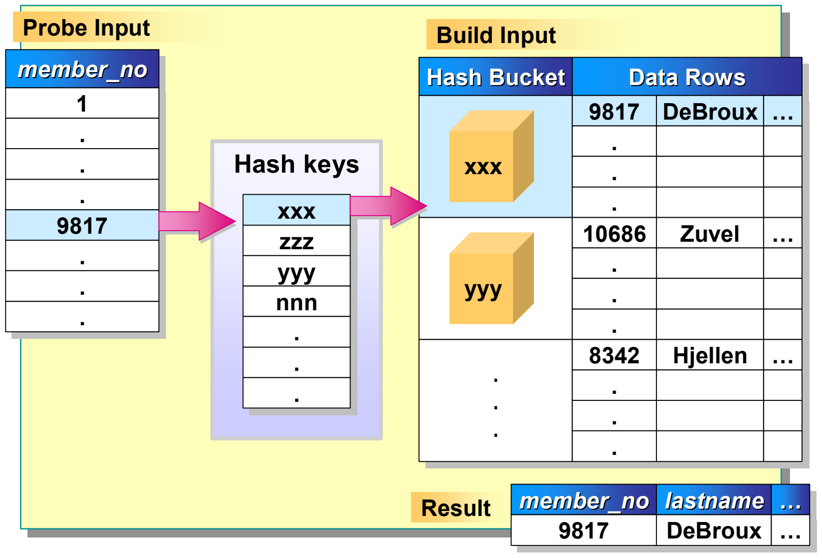
  If the merge join is a many-to-many operation, SQL Server uses a temporary table to store the rows. If duplicate values from each input exist, one of the inputs rewinds to the start of the duplicates as each duplicate value from the other input is processed.
- Query performance for a merge join is very fast, but the cost can be high if the query optimizer must perform its own sort operation.

  If the data volume is large and the desired data can be obtained presorted from existing Balanced-Tree (B-Tree) indexes, merge join is often the fastest join algorithm.
- A merge join is typically used if the two join inputs have a large amount of data and are sorted on their join columns (for example, if the join inputs were obtained by scanning sorted indexes).
- Merge join operations can only be performed with an equality operator in the join predicate.

How Hash Joins Are Processed

Topic Objective
To discuss how hash joins are processed.

Lead-in
The query optimizer uses a hash join when it estimates that this option is more efficient than processing queries by using a nested loop or merge join.



Hashing is a strategy for dividing data into equal sets of a manageable size based on a given property or characteristic. The grouped data can then be used to determine whether a particular data item matches an existing value.

Note Duplicate data or ranges of data are not useful for hash joins because the data is not organized together or in order.

When a Hash Is Join Used

The query optimizer uses a hash join option when it estimates that it is more efficient than processing queries by using a nested loop or merge join. It typically uses a hash join when an index does not exist or when existing indexes are not useful.

Assigns a Build and Probe Input

The query optimizer assigns a *build* and *probe input*. If the query optimizer incorrectly assigns the build and probe input (this may occur because of imprecise density estimates), it reverses them dynamically. The ability to change input roles dynamically is called *role reversal*.

Build input consists of the column values from a table with the lowest number of rows. Build input creates a *hash table* in memory to store these values.

Hash bucket is a storage place in the hash table in which each row of the build input is inserted. Rows from one of the join tables are placed into the hash bucket where the hash key value of the row matches the hash key value of the bucket. Hash buckets are stored as a linked list and only contain the columns that are needed for the query.

A hash table contains hash buckets. The hash table is created from the build input.

Probe input consists of the column values from the table with the most rows. Probe input is what the build input checks to find a match in the hash buckets.

Note The query optimizer uses column or index statistics to help determine which input is the smaller of the two.

Processing a Hash Join

The following list is a simplified description of how the query optimizer processes a hash join. It is not intended to be comprehensive because the algorithm is very complex. SQL Server:

1. Reads the probe input. Each probe input is processed one row at a time.
2. Performs the hash algorithm against each probe input and generates a hash key value.
3. Finds the hash bucket that matches the hash key value.
4. Accesses the hash bucket and looks for the matching row.
5. Returns the row if a match is found.

Performance Considerations

Consider the following facts about the hash joins that the query optimizer uses:

- Similar to merge joins, a hash join is very efficient, because it uses hash buckets, which are like a dynamic index but with less overhead for combining rows.
- Hash joins can be performed for all types of join operations (except cross join operations), including UNION and DIFFERENCE operations.
- A hash operator can remove duplicates and group data, such as SUM (salary) GROUP BY department. The query optimizer uses only one input for both the build and probe roles.
- If join inputs are large and are of similar size, the performance of a hash join operation is similar to a merge join with prior sorting. However, if the size of the join inputs is significantly different, the performance of a hash join is often much faster.
- Hash joins can process large, unsorted, non-indexed inputs efficiently. Hash joins are useful in complex queries because the intermediate results:
  - Are not indexed (unless explicitly saved to disk and then indexed).
  - Are often not sorted for the next operation in the execution plan.

- The query optimizer can identify incorrect estimates and make corrections dynamically to process the query more efficiently.
- A hash join reduces the need for database *denormalization*. Denormalization is typically used to achieve better performance by reducing join operations despite redundancy, such as inconsistent updates. Hash joins give you the option to vertically partition your data as part of your physical database design. Vertical partitioning represents groups of columns from a single table in separate files or indexes.

Note For additional information on hash joins, search on "understanding hash joins" in SQL Server Books Online.

Lab B: Analyzing Queries That Use Different Join Strategies

Topic Objective
To introduce the lab.

Lead-in
In this lab, you will observe the execution plan and evaluate how the query optimization occurred by using different join strategies.

Explain the lab objectives.

Objectives

After completing this lab, you will be able to evaluate how the query optimizer processes a query by using nested loop, merge, and hash join strategies.

Prerequisites

Before working on this lab, you must have:

- Script files, which are located in C:\Moc\2073A\Labfiles\L14.

Lab Setup

To complete this lab, you must have either:

- Completed the prior lab, or
- Executed the C:\Moc\2073A\Batches\Restore14B.cmd batch file.

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

For More Information

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

Other resources that you can use include:

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

Scenario

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

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

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

Estimated time to complete this lab: 30 minutes

Exercise 1
Processing Nested Loop Joins

In this exercise, you will create indexes on the **member** and **charge** tables and observe how the query optimizer processes the query by using a nested loop join strategy.

You can open, review, and execute sections of the NestedLoopJoin.sql script file in C:\Moc\2073A\Labfiles\L14, or type and execute the provided Transact-SQL statements.

► To create indexes

In this procedure, you will create indexes on the **member** and **charge** tables.

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

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

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

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

3. With SQL Query Analyzer, type and execute this statement to drop existing indexes on the **member** and **charge** tables:

```
USE credit
EXEC index_cleanup member
EXEC index_cleanup charge
```

4. Type and execute this statement to create a unique, nonclustered composite index on the **lastname** and **firstname** columns of the **member** table:

```
USE credit
CREATE UNIQUE nonclustered INDEX mbr_name
   ON member(lastname, firstname)
```

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

```
USE credit
CREATE nonclustered INDEX chg_mem_no
   ON charge(member_no)
```

► **To observe how a query is processed by using a nested loop join strategy**

In this procedure, you will set the statistics option to ON, execute a query, and record the statistical information.

1. Type and execute this statement to set the statistics option to ON:

   ```
   SET STATISTICS IO ON
   ```

2. In the Query window, on the **Query** menu, click **Show Execution Plan**.
3. Type and execute this SELECT statement to retrieve member number, last name, and charge number for members with last name Barr and first name Bos.

   ```
   USE credit
   SELECT member.member_no, lastname, charge_no
      FROM member JOIN charge
      ON member.member_no = charge.member_no
      WHERE member.lastname = 'BARR'
          AND firstname = 'BOS'
   ```

4. Record the statistical information for the **member** table.

Information	Result
Scan count	**1**
Number of logical reads	**3**
Number and name of indexes used to process the query	**1 (mbr_name)**

5. Record the statistical information for the **charge** table.

Information	Result
Scan count	**1**
Number of logical reads	**26**
Number and name of indexes used to process the query	**1 (chg_mem_no)**

   Why does the **member** table have three I/O, whereas the **charge** table requires 26 I/O?

   For one row in the member table, there are multiple rows in the charge table. This shows that you have a standard one-to-many relationship.

6. Click the **Execution Plan** tab to display the execution plan graphically.
7. Examine the execution plan.

   What strategy did the query optimizer use to find the rows in both tables?

   Nested loop/inner join. The query optimizer used the nonclustered index to find the member name BOS BARR in the member table. Having found the correct member, it used the member_no column to look for matching rows in the chg_mem_no index of the charge table. This is the nested loop/inner join.

Exercise 2
Processing Merge Joins

In this exercise, you will drop all indexes on the **member** and **charge** tables, execute a query, and evaluate the execution plan. Then, you will create indexes on the **member** and **charge** tables, re-execute a query, and compare page I/O.

You can open, review, and execute sections of the MergeJoin.sql script file in C:\Moc\2073A\Labfiles\L14, or type and execute the provided Transact-SQL statements.

► **To observe how the query optimizer processes a query against a table with no indexes**

In this procedure, you will drop all indexes on the **member** and **charge** tables, execute a query, and record the statistical information.

1. Type and execute these statements to drop existing indexes on the **member** and **charge** tables:

```
USE credit
EXEC index_cleanup member
EXEC index_cleanup charge
```

2. With the STATISTICS IO option set to ON, type and execute this SELECT statement to retrieve member number, last name, and charge number for members with the last name Hahn:

```
USE credit
   SELECT member.member_no, lastname, charge_no
   FROM member JOIN charge
   ON member.member_no = charge.member_no
   WHERE member.lastname = 'HAHN'
```

Note This query is referred to as the original query throughout this exercise.

3. Record the statistical information for the **member** table.

Information	Result
Scan count	**1**
Number of logical reads	**142**
Number and name of indexes used to process the query	**None (table scan)**

4. Record the statistical information for the **charge** table.

Information	Result
Scan count	**1**
Number of logical reads	**582**
Number and name of indexes used to process the query	**None (table scan)**

► To create indexes and evaluate how the query optimizer processes a query against a table with indexes

In this procedure, you will create indexes on the **member** and **charge** tables, re-execute the original query, and evaluate how the query optimizer processed the query by using a merge join strategy.

1. Type and execute this statement to create a nonclustered, composite index on the **member_no** and **lastname** columns of the **member** table:

```
USE credit
CREATE nonclustered INDEX mbr_name
   ON member(member_no, lastname)
```

2. Type and execute this statement to create a nonclustered, composite index on the **member_no** and **charge_no** columns of the **charge** table:

```
USE credit
CREATE nonclustered INDEX chg_charge_no
   ON charge(member_no, charge_no)
```

3. Re-execute the original query, which retrieves member number, last name, and charge number for members with the last name Hahn.

Original Query

```
USE credit
SELECT member.member_no, lastname, charge_no
   FROM member JOIN charge
   ON member.member_no = charge.member_no
      WHERE member.lastname = 'HAHN'
```

4. Record the statistical information for the **member** table.

Information	Result
Scan count	**1**
Number of logical reads	**32**
Number and name of indexes used to process the query	**1 (mbr_name)**

5. Record the statistical information for the **charge** table.

Information	Result
Scan count	**1**
Number of logical reads	**158**
Number and name of indexes used to process the query	**1 (chg_charge_no)**

6. Compare the statistical information for a query executed against a table with no indexes, and the same query executed against a table with useful indexes.

   Was I/O reduced by adding indexes?

   Yes. By using the indexes, the query optimizer reads only the pages required to return the results.

   ____

   ____

7. Click the **Execution Plan** tab to display the execution plan graphically.
8. Examine the execution plan.

   What join strategy did the query optimizer use to process the join?

   Merge join/inner join. The query optimizer scanned portions of the index on the charge table and portions of the index on the member table. It was able to join (merge) that information and return the requested information.

   ____

   ____

   ____

   ____

   Why did the query optimizer select this strategy?

   The query optimizer selected this strategy because it was able to cover the query for each table. Because it covered the query, it only scanned the leaf-level pages of the nonclustered indexes, which are maintained in sorted order. This allowed the merge join to take advantage of inputs from each table, which where already sorted.

   ____

   ____

   ____

   ____

Exercise 3
Processing Hash Joins

In this exercise, you will drop all indexes on the **member** and **charge** tables, execute a query, and observe how the query optimizer processes the query by using a hash join strategy.

You can open, review, and execute sections of the HashJoin.sql script file in C:\Moc\2073A\Labfiles\L14, or type and execute the provided Transact-SQL statements.

► To observe how a query is processed by using a hash join strategy

In this procedure, you will drop all indexes on the **member** and **charge** tables, execute a query, and evaluate how the query optimizer processed the query.

1. Type and execute these statements to drop existing indexes on the **member** and **charge** tables:

   ```
   USE credit
   EXEC index_cleanup member
   EXEC index_cleanup charge
   ```

2. Type and execute this SELECT statement to retrieve member number, last name, and charge number for members with the last name Barr and the first name Bos:

   ```
   USE credit
   SELECT m.member_no, lastname, charge_no
      FROM member m JOIN charge c
      ON m.member_no = c.member_no
      WHERE m.lastname = 'BARR'
      AND firstname = 'BOS'
   ```

3. Click the **Execution Plan** tab to display the execution plan graphically.
4. Examine the execution plan.

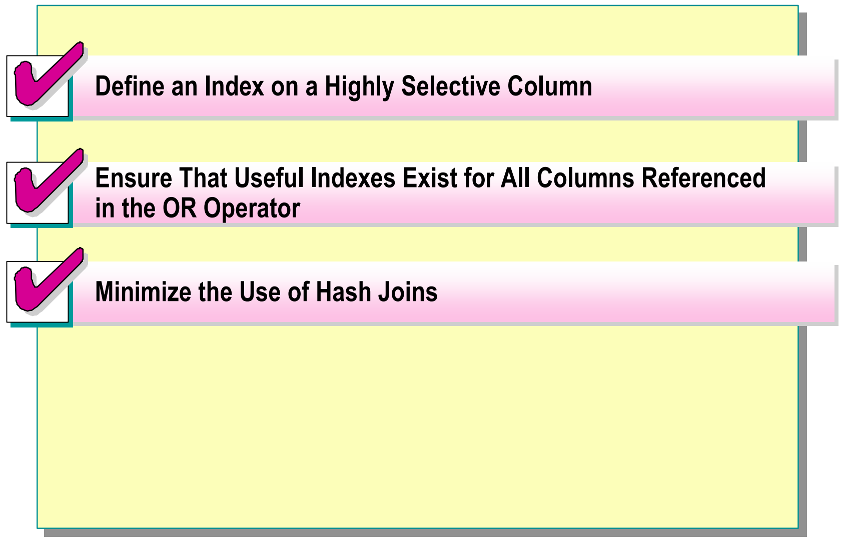
   What strategy did the query optimizer use to find these rows? Why?

   Hash match/inner join. Without indexes, the query optimizer must scan both tables to find the requested rows. Then, the query optimizer builds a hash table from qualifying rows of the member table and processes each row of the charge table, returning matching rows.

Recommended Practices

Topic Objective
To list the recommended practices when analyzing queries that use the AND and OR operators or join operations.

Lead-in
When analyzing queries that use the AND and OR operators or join operations, you should consider the following practices.



✓ Define an Index on a Highly Selective Column

✓ Ensure That Useful Indexes Exist for All Columns Referenced in the OR Operator

✓ Minimize the Use of Hash Joins

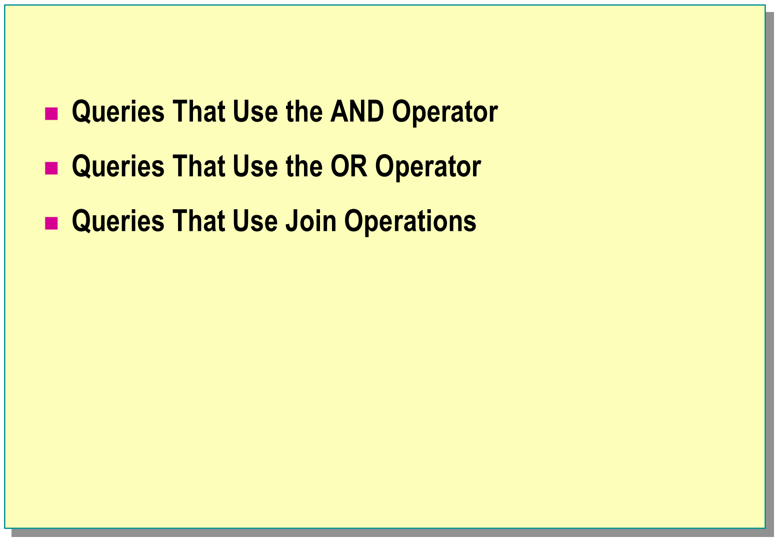
When analyzing queries that use the AND and OR operators or join operations, you should consider the following practices:

- Define an index on a highly selective column. The best way to index for queries that contain the AND operator is to have at least one highly selective search criterion, and define an index on that column.
- Ensure that useful indexes exist for all columns referenced in the OR operator.Minimize the use of hash joins by creating useful indexes and writing efficient queries.

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.



- **Queries That Use the AND Operator**
- **Queries That Use the OR Operator**
- **Queries That Use Join Operations**

1. You are writing queries for an application. You are not sure about the benefits of using multiple restrictions in the WHERE clause by using the AND operator. What are some of the advantages of using multiple AND operators in your queries?

   The more AND operators that you use, the more restrictive the query becomes. Using more AND operators also allows the query to potentially use many indexes, or it offers a better choice of indexes.

2. A query is performing poorly. The query optimizer currently performs a table scan even though indexes exist on some of the columns referenced in the WHERE clause. What could be causing the poor performance of the following query?

```
SELECT * FROM member
   WHERE lastname = 'GOHAN'
   OR expr_dt < '12/31/1999'
   OR region_no = 7
```

   When using the OR operator in a query, it is necessary that every column using the OR operator have an index or a useful index. If just one column has no index or no useful index, then the query optimizer performs a table scan.

3. One of your queries is performing adequately, but you would like to determine whether you could improve its performance. Currently, the query optimizer performs a hash join operation. What can be done to possibly improve performance?

   A hash join is not necessarily bad. To improve performance, verify that the join columns have indexes or have useful indexes. You can also verify that the query contains a WHERE clause, verify that the search condition limits the search, and verify that useful indexes exist on the columns referenced in the WHERE clause.