

Execution Time

Both Duration and CPU represent the time factor of a query. To obtain detailed information on the amount of time (in milliseconds) required to parse, compile, and execute a query, use SET STATISTICS TIME as follows:

```
SET STATISTICS TIME ON;
GO
SELECT soh.AccountNumber,
       sod.LineTotal,
       sod.OrderQty,
       sod.UnitPrice,
       p.Name
FROM Sales.SalesOrderHeader soh
     JOIN Sales.SalesOrderDetail sod
         ON soh.SalesOrderID = sod.SalesOrderID
     JOIN Production.Product p
         ON sod.ProductID = p.ProductID
WHERE sod.LineTotal > 1000;
GO
SET STATISTICS TIME OFF;
GO
```

The output of STATISTICS TIME for the preceding SELECT statement is shown here:

```
SQL Server parse and compile time:
  CPU time = 0 ms, elapsed time = 9 ms.

(32101 row(s) affected)
```

```
SQL Server Execution Times:
  CPU time = 156 ms, elapsed time = 400 ms.
SQL Server parse and compile time:
  CPU time = 0 ms, elapsed time = 0 ms.
```

The CPU time = 156 ms part of the execution times represents the CPU value provided by Extended Events. Similarly, the corresponding Elapsed time = 400 ms represents the Duration value provided by the other mechanisms.

A 0 ms parse and 9 ms compile time signifies that the optimizer has to parse the query first for syntax and then compile it to produce the execution plan.

STATISTICS IO

As discussed in the “Identifying Costly Queries” section earlier in the chapter, the number of reads in the Reads column is frequently the most significant cost factor among duration, cpu, reads, and writes. The total number of reads performed by a query consists of the sum of the number of reads performed on all tables involved in the query. The reads performed on the individual tables may vary significantly, depending on the size of the result set requested from the individual table and the indexes available.

To reduce the total number of reads, it will be useful to find all the tables accessed in the query and their corresponding number of reads. This detailed information helps you concentrate on optimizing data access on the tables with a large number of reads. The number of reads per table also helps you evaluate the impact of the optimization step (implemented for one table) on the other tables referred to in the query.

In a simple query, you determine the individual tables accessed by taking a close look at the query. This becomes increasingly difficult the more complex the query becomes. In the case of stored procedures, database views, or functions, it becomes more difficult to identify all the tables actually accessed by the optimizer. You can use `STATISTICS IO` to get this information, irrespective of query complexity.

To turn `STATISTICS IO` on, navigate to Query ► Query Options ► Advanced ► Set Statistics IO in Management Studio. You may also get this information programmatically as follows:

```
SET STATISTICS IO ON;
GO
SELECT soh.AccountNumber,
       sod.LineTotal,
       sod.OrderQty,
       sod.UnitPrice,
       p.Name
FROM Sales.SalesOrderHeader soh
     JOIN Sales.SalesOrderDetail sod
       ON soh.SalesOrderID = sod.SalesOrderID
```

```

JOIN Production.Product p
  ON sod.ProductID = p.ProductID
WHERE sod.SalesOrderID = 71856;
GO
SET STATISTICS IO OFF;
GO

```

If you run this query and look at the execution plan, it consists of three clustered index seeks with two loop joins. If you remove the WHERE clause and run the query again, you get a set of scans and some hash joins. That's an interesting fact—but you don't know how it affects the query I/O usage! You can use SET STATISTICS IO as shown previously to compare the cost of the query (in terms of logical reads) between the two processing strategies used by the optimizer.

You get following STATISTICS IO output when the query uses the hash join:

```

(121317 row(s) affected)
Table 'Workfile'. Scan count 0, logical reads 0...
Table 'Worktable'. Scan count 0, logical reads 0...
Table 'SalesOrderDetail'. Scan count 1, logical reads 1248...
Table 'SalesOrderHeader'. Scan count 1, logical reads 689...
Table 'Product'. Scan count 1, logical reads 6...

(1 row(s) affected)

```

Now when you add back in the WHERE clause to appropriately filter the data, the resultant STATISTICS IO output turns out to be this:

```

(2 row(s) affected)
Table 'Product'. Scan count 0, logical reads 4...
Table 'SalesOrderDetail'. Scan count 1, logical reads 3...
Table 'SalesOrderHeader'. Scan count 0, logical reads 3...

(1 row(s) affected)

```

Logical reads for the SalesOrderDetail table have been cut from 1,248 to 3 because of the index seek and the loop join. It also hasn't significantly affected the data retrieval cost of the Product table.

While interpreting the output of `STATISTICS IO`, you mostly refer to the number of logical reads. The number of physical reads and read-ahead reads will be nonzero when the data is not found in the memory, but once the data is populated in memory, the physical reads and read-ahead reads will tend to be zero.

There is another advantage to knowing all the tables used and their corresponding reads for a query. Both the `duration` and `CPU` values may fluctuate significantly when reexecuting the same query with no change in table schema (including indexes) or data because the essential services and background applications running on the SQL Server machine can affect the processing time of the query under observation. But, don't forget that logical reads are not always the most accurate measure. `Duration` and `CPU` are absolutely useful and an important part of any query tuning.

During optimization steps, you need a nonfluctuating cost figure as a reference. The reads (or logical reads) don't vary between multiple executions of a query with a fixed table schema and data. For example, if you execute the previous `SELECT` statement ten times, you will probably get ten different figures for `duration` and `CPU`, but `Reads` will remain the same each time. Therefore, during optimization, you can refer to the number of reads for an individual table to ensure that you really have reduced the data access cost of the table. Just never assume that is your only measure or even the primary one. It's just a constant measure and therefore useful.

Even though the number of logical reads can also be obtained from Extended Events, you get another benefit when using `STATISTICS IO`. The number of logical reads for a query shown by Profiler or the Server Trace option increases as you use different `SET` statements (mentioned previously) along with the query. But the number of logical reads shown by `STATISTICS IO` doesn't include the additional pages that are accessed because `SET` statements are used with a query. Thus, `STATISTICS IO` provides a consistent figure for the number of logical reads.

Actual Execution Plans

As mentioned earlier in the chapter, actual execution plans now capture and display some query performance metrics within the execution plan itself along with the traditional metrics. If we open the `SELECT` operator for the previous query and plan, the one without the `WHERE` clause, Figure 7-26 shows both the `QueryTimeStats` and `WaitStats` values.

[-] QueryTimeStats	
CpuTime	565
ElapsedTime	1434
RetrievedFromCache	true
SecurityPolicyApplied	False
[+] Set Options	ANSI_NULLS: True, ANSI_P
Statement	SELECT soh.AccountNumI
[-] WaitStats	
WaitCount	1979
WaitTimeMs	876
WaitType	ASYNC_NETWORK_IO

Figure 7-26. *QueryTimeStats and WaitStats within an actual execution plan*

You can now see the CpuTime and ElapsedTime for the query directly within the execution plan, as long as you’re capturing an actual execution plan. These values are measured in milliseconds. You can also see the top wait or waits for a query. In the Figure 7-26 example it’s ASYNC_NETWORK_IO, probably explained by the fact that we’re returning 121,000 rows across the network. The wait statistics show up only if they are longer than 1ms. This does lead to the waits shown within an execution being not as accurate as the other mechanisms for capturing waits. However, this is a handy tool to help evaluate the behavior of the query within the execution plan.

This gives you yet another quick and easy way to see query performance. If you look at one of the other operators, you can also see the I/O for that operator, measured in pages.

Summary

In this chapter, you saw that you can use Extended Events to identify the queries causing a high amount of stress on the system resources in a SQL workload. Collecting the session data can, and should be, automated using system stored procedures. For immediate access to statistics about running queries, use the DMV `sys.dm_exec_query_stats`. You can further analyze these queries with Management Studio to find the costly steps in the processing strategy of the query. For better performance, it is important to consider both the index and join mechanisms used in an execution plan while analyzing a query. The number of data retrievals (or reads) for the individual tables provided by

SET STATISTICS IO helps concentrate on the data access mechanism of the tables with the most reads. You also should focus on the CPU cost and overall time of the most costly queries.

Once you identify a costly query and finish the initial analysis, the next step should be to optimize the query for performance. Because indexing is one of the most commonly used performance-tuning techniques, in the next chapter. I will discuss in depth the various indexing mechanisms available in SQL Server.

CHAPTER 8

Index Architecture and Behavior

The right index on the right column, or columns, is the basis on which query tuning begins. A missing index or an index placed on the wrong column, or columns, can be the basis for all performance problems starting with basic data access, continuing through joins, and ending in filtering clauses. For these reasons, it is extremely important for everyone—not just a DBA—to understand the different indexing techniques that can be used to optimize the database design.

In this chapter, I cover the following topics:

- What an index is
- The benefits and overhead of an index
- General recommendations for index design
- Clustered and nonclustered index behavior and comparisons
- Recommendations for clustered and nonclustered indexes

What Is an Index?

One of the best ways to reduce disk I/O is to use an index. An index allows SQL Server to find data in a table without scanning the entire table. An index in a database is analogous to an index in a book. Say, for example, that you wanted to look up the phrase *table scan* in this book. In the paper version, without the index at the back of the book, you would have to peruse the entire book to find the text you needed. With the index, you know exactly where the information you want is stored.

While tuning a database for performance, you create indexes on the different columns used in a query to help SQL Server find data quickly. For example, the following query against the `Production.Product` table results in the data shown in Figure 8-1 (the first 10 of 500+ rows):

```
SELECT TOP 10
    p.ProductID,
    p.[Name],
    p.StandardCost,
    p.[Weight],
    ROW_NUMBER() OVER (ORDER BY p.Name DESC) AS RowNumber
FROM Production.Product p
ORDER BY p.Name DESC;
```

	ProductID	Name	StandardCost	Weight	RowNumber
1	852	Women's Tights, S	30.9334	NULL	1
2	853	Women's Tights, M	30.9334	NULL	2
3	854	Women's Tights, L	30.9334	NULL	3
4	867	Women's Mountain Shorts, S	26.1763	NULL	4
5	868	Women's Mountain Shorts, M	26.1763	NULL	5
6	869	Women's Mountain Shorts, L	26.1763	NULL	6
7	870	Water Bottle - 30 oz.	1.8663	NULL	7
8	842	Touring-Panniers, Large	51.5625	NULL	8
9	965	Touring-3000 Yellow, 62	461.4448	30.00	9
10	964	Touring-3000 Yellow, 58	461.4448	29.79	10

Figure 8-1. Sample `Production.Product` table

The preceding query scanned the entire table since there was no `WHERE` clause. If you need to add a filter through the `WHERE` clause to retrieve all the products where `StandardCost` is greater than 150, without an index the table will still have to be scanned, checking the value of `StandardCost` at each row to determine which rows contain a value greater than 150. An index on the `StandardCost` column could speed up this process by providing a mechanism that allows a structured search against the data rather than a row-by-row check. You can take two different, and fundamental, approaches for creating this index.

- *Like a dictionary:* A dictionary is a distinct listing of words in alphabetical order. An index can be stored in a similar fashion. The data is ordered, although it will still have duplicates. The first ten rows, ordered by `StandardCost DESC` instead of by `Name`, would look like the data shown in Figure 8-2. Notice the `RowNumber` column shows the original placement of the row when ordering by `Name`.

	ProductID	Name	StandardCost	Weight	RowNumber
1	749	Road-150 Red, 62	2171.2942	15.00	125
2	753	Road-150 Red, 56	2171.2942	14.68	126
3	752	Road-150 Red, 52	2171.2942	14.42	127
4	751	Road-150 Red, 48	2171.2942	14.13	128
5	750	Road-150 Red, 44	2171.2942	13.77	129
6	774	Mountain-100 Silver, 48	1912.1544	21.42	170
7	773	Mountain-100 Silver, 44	1912.1544	21.13	171
8	772	Mountain-100 Silver, 42	1912.1544	20.77	172
9	771	Mountain-100 Silver, 38	1912.1544	20.35	173
10	778	Mountain-100 Black, 48	1898.0944	21.42	174

Figure 8-2. *Product table sorted on StandardCost*

So, now if you wanted to find all the data in the rows where `StandardCost` is greater than 150, the index would allow you to find them immediately by moving down to the first value greater than 150. An index that applies order to the data stored based on the index key order is known as a *clustered index*. Because of how SQL Server stores data, this is one of the most important indexes in your database design. I explain this in detail later in the chapter.

- *Like a book's index architecture:* An ordered list can be created without altering the layout of the table, similar to the way the index of a book is created. Just like the keyword index of a book lists the keywords in a separate section with a page number to refer to the main content of the book, the list of `StandardCost` values is created as a separate structure and refers to the corresponding row in the `Product` table through a pointer. For the example, I'll use `RowNumber` as the pointer. Table 8-1 shows the structure of the manufacturer index.

Table 8-1. *Structure of the Manufacturer Index*

StandardCost	RowNumber
2171.2942	125
2171.2942	126
2171.2942	127
2171.2942	128
2171.2942	129
1912.1544	170

SQL Server can scan the manufacturer index to find rows where StandardCost is greater than 150. Since the StandardCost values are arranged in a sorted order, SQL Server can stop scanning as soon as it encounters the row with a value of 150 or less. This type of index is called a *nonclustered index*, and I explain it in detail later in the chapter.

In either case, SQL Server will be able to find all the products where StandardCost is greater than 150 more quickly than without an index under most circumstances.

You can create indexes on either a single column (as described previously) or a combination of columns in a table. SQL Server also automatically creates indexes for certain types of constraints (for example, PRIMARY KEY and UNIQUE constraints).

The Benefit of Indexes

SQL Server has to be able to find data, even when no index is present on a table. When no clustered index is present to establish a storage order for the data, the storage engine will simply read through the entire table to find what it needs. A table without a clustered index is called a *heap table*. A heap is just an unordered stack of data with a row identifier as a pointer to the storage location. This data is not ordered or searchable except by walking through the data, row by row, in a process called a *scan*. When a clustered index is placed on a table, the key values of the index establish an order for the data. Further, with a clustered index, the data is stored with the index so that the data itself is now ordered. When a clustered index is present, the pointer on the nonclustered index consists of the values that define the clustered index key. This is a big part of what makes clustered indexes so important.

Data within SQL Server is stored on a page, which is 8KB in size. A page is the minimum amount of information that moves off the disk and into memory, so how much you can store on a page becomes important. Since a page has a limited amount of space, it can store a larger number of rows if the rows contain a fewer number of columns or the columns are of smaller size. The nonclustered index usually doesn't (and shouldn't) contain all the columns of the table; it usually contains only a limited number of the columns. Therefore, a page will be able to store more rows of a nonclustered index than rows of the table itself, which contains all the columns. Consequently, SQL Server will be able to read more values for a column from a page representing a nonclustered index on the column than from a page representing the table that contains the column.

Another benefit of a nonclustered index is that because it is in a separate structure from the data table, it can be put in a different filegroup, with a different I/O path, as explained in Chapter 3. This means SQL Server can access the index and table concurrently, making searches even faster.

Indexes store their information in a balanced tree, referred to as a *B-tree*, structure, so the number of reads required to find a particular row is minimized. The following example shows the benefit of a B-tree structure.

Consider a single-column table with 27 rows in a random order and only 3 rows per leaf page. Suppose the layout of the rows in the pages is as shown in Figure 8-3.

24,14,12	11,20,9	25,15,10	16,13,7	2,26,17	21,18,22	19,6,5	1,8,3	27,4,23
----------	---------	----------	---------	---------	----------	--------	-------	---------

Figure 8-3. Initial layout of 27 rows

To search the row (or rows) for the column value of 5, SQL Server has to scan all the rows and the pages since even the last row in the last page may have the value 5. Because the number of reads depends on the number of pages accessed, nine read operations (retrieving pages from the disk and transferring them to memory) have to be performed without an index on the column. This content can be ordered by creating an index on the column, with the resultant layout of the rows and pages shown in Figure 8-4.

1,2,3	4,5,6	7,8,9	10,11,12	13,14,15	16,17,18	19,20,21	22,23,24	25,26,27
-------	-------	-------	----------	----------	----------	----------	----------	----------

Figure 8-4. Ordered layout of 27 rows

Indexing the column arranges the content in a sorted fashion. This allows SQL Server to determine the possible value for a row position in the column with respect to the value of another row position in the column. For example, in Figure 8-4, when SQL Server finds the first row with the column value 6, it can be sure that there are no more rows with the column value 5. Thus, only two read operations are required to fetch the rows with the value 5 when the content is indexed. However, what happens if you want to search for the column value 25? This will require nine read operations! This problem is solved by implementing indexes using the B-tree structure (as shown in Figure 8-5).

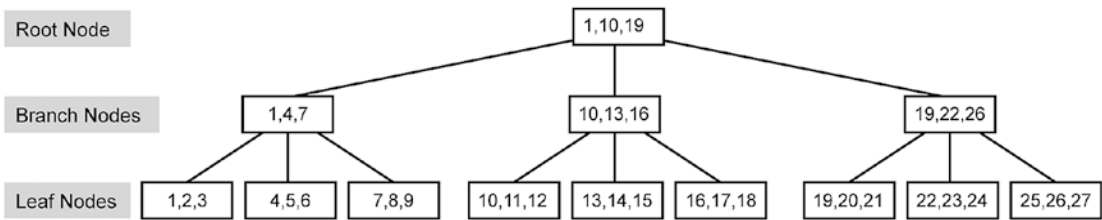


Figure 8-5. B-tree layout of 27 rows

A B-tree consists of a starting node (or page) called a *root node* with *branch nodes* (or pages) growing out of it (or linked to it). All keys are stored in the leaves. Contained in each interior node (above the leaf nodes) are pointers to its branch nodes and values representing the smallest value found in the branch node. Keys are kept in sorted order within each node. B-trees use a balanced tree structure for efficient record retrieval—a B-tree is balanced when the leaf nodes are all at the same level from the root node. For example, creating an index on the preceding content will generate the balanced B-tree structure shown in Figure 8-5. At the bottom level, all the leaf nodes are connected to each other through a doubly linked list, meaning each page points to the page that follows it, and the page that follows it points back to the preceding page. This prevents having to go back up the chain when pages are traversed beyond the definitions of the intermediate pages.

The B-tree algorithm minimizes the number of pages to be accessed to locate a desired key, thereby speeding up the data access process. For example, in Figure 8-5, the search for the key value 5 starts at the top root node. Since the key value is between 1 and 10, the search process follows the left branch to the next node. As the key value 5 falls between the values 4 and 7, the search process follows the middle branch to the next node with the starting key value of 4. The search process retrieves the key value 5 from this leaf page. If the key value 5 doesn't exist in this page, the search process will stop since it's the leaf page. Similarly, the key value 25 can also be searched using the same number of reads.

Index Overhead

The performance benefit of indexes does come at a cost. Tables with indexes require more storage and memory space for the index pages in addition to the data pages of the table. Data manipulation queries (INSERT, UPDATE, and DELETE statements, or the CRUD part of Create, Read, Update, Delete [CRUD]) can take longer, and more processing time is required to maintain the indexes of constantly changing tables. This is because, unlike a SELECT statement, data manipulation queries modify the data content of a table. If an INSERT statement adds a row to the table, then it also has to add a row in the index structure. If the index is a clustered index, the overhead is greater still because the row has to be added to the data pages themselves in the right order, which may require other data rows to be repositioned below the entry position of the new row. The UPDATE and DELETE data manipulation queries change the index pages in a similar manner.

When designing indexes, you'll be operating from two different points of view: the existing system, already in production, where you need to measure the overall impact of an index, and the tactical approach where all you worry about is the immediate benefits of an index, usually when initially designing a system. When you have to deal with the existing system, you should ensure that the performance benefits of an index outweigh the extra cost in processing resources. You can do this by using Extended Events (explained in Chapter 3) to do an overall workload optimization (explained in Chapter 27). When you're focused exclusively on the immediate benefits of an index, SQL Server supplies a series of dynamic management views that provide detailed information about the performance of indexes, `sys.dm_db_index_operational_stats` or `sys.dm_db_index_usage_stats`. The view `sys.dm_db_index_operational_stats` shows the low-level activity, such as locks and I/O, on an index that is in use. The view `sys.dm_db_index_usage_stats` returns statistical counts of the various index operations that have occurred to an index over time. Both of these will be used more extensively in Chapter 21 when I discuss blocking.

Note In some parts of the book, I use the `STATISTICS IO` and `STATISTICS TIME` measurements against the queries that I'm running. You can add `SET` commands to the code, or you can change the connection settings for the query window. I suggest just changing the connection settings. However, there should also be a warning here. Using `STATISTICS IO` and `STATISTICS TIME` together can sometimes cause problems. The time it takes to retrieve the I/O information

is added to the STATISTICS TIME information, thus skewing the results. If you don't need table-level I/O, it's better to capture the execution metrics using Extended Events. You also can get CpuTime and ElapsedTime from actual execution plans if you're capturing one for a query.

To understand the overhead cost of an index on data manipulation queries, consider the following example. First, create a test table with 10,000 rows.

```
DROP TABLE IF EXISTS dbo.Test1;
GO
CREATE TABLE dbo.Test1
(
    C1 INT,
    C2 INT,
    C3 VARCHAR(50)
);

WITH Nums
AS (SELECT TOP (10000)
    ROW_NUMBER() OVER (ORDER BY (SELECT 1)) AS n
    FROM master.sys.all_columns ac1
    CROSS JOIN master.sys.all_columns ac2
)
INSERT INTO dbo.Test1
(
    C1,
    C2,
    C3
)
SELECT n,
    n,
    'C3'
FROM Nums;
```

Run an UPDATE statement, like so:

```
UPDATE dbo.Test1
SET C1 = 1,
    C2 = 1
WHERE C2 = 1;
```

Then the number of logical reads reported by SET STATISTICS IO is as follows:

Table 'Test1'. Scan count 1, logical reads 29

Add an index on column c1, like so:

```
CREATE CLUSTERED INDEX iTest
ON dbo.Test1(C1);
```

Then the resultant number of logical reads for the same UPDATE statement increases from 29 to 38 but also has added a worktable with an additional 5 reads, for a total of 43.

Table 'Test1'. Scan count 1, logical reads 38

Table 'Worktable'. Scan count 1, logical reads 5

The number of reads goes up because it was necessary to rearrange the data in order to store it in the correct order within the clustered index, increasing the number of reads beyond what was necessary for a heap table to just add the data to the end of the existing storage.

Note A *worktable* is a temporary table used internally by SQL Server to process the intermediate results of a query. Worktables are created in the tempdb database and are dropped automatically after query execution.

Even though it is true that the amount of overhead required to maintain indexes increases for data manipulation queries, be aware that SQL Server must first find a row before it can update or delete it; therefore, indexes can be helpful for UPDATE and DELETE statements with necessary WHERE clauses. The increased efficiency in using the index to locate a row usually offsets the extra overhead needed to update the indexes, unless the table has a lot of indexes or lots of updates. Further, the vast majority of systems are read heavy, meaning they have a lot more data being retrieved than is being inserted or modified.

To understand how an index can benefit even data modification queries, let's build on the example. Create another index on table Test1. This time, create the index on column C2 referred to in the WHERE clause of the UPDATE statement.

```
CREATE NONCLUSTERED INDEX iTest2
ON dbo.Test1(C2);
```

After adding this new index, run the UPDATE command again.

```
UPDATE  dbo.Test1
SET      C1 = 1,
          C2 = 1
WHERE    C2 = 1;
```

The total number of logical reads for this UPDATE statement decreases from 43 to 20 (= 15 + 5).

Table 'Test1'. Scan count 1, logical reads 15

Table 'Worktable'. Scan count 1, logical reads 5

The examples in this section have demonstrated that although having an index adds some overhead cost to action queries, the overall result can be a decrease in cost because of the beneficial effect of indexes on searching, even during updates.

Index Design Recommendations

The main recommendations for index design are as follows:

- Examine the WHERE clause and JOIN criteria columns.
- Use narrow indexes.
- Examine column uniqueness and selectivity.
- Examine the column data type.
- Consider column order.
- Consider the type of index (clustered versus nonclustered).

Let's consider each of these recommendations in turn.

Examine the WHERE Clause and JOIN Criteria Columns

When a query is submitted to SQL Server, the query optimizer tries to find the best data access mechanism for every table referred to in the query. Here is how it does this:

1. The optimizer identifies the columns included in the WHERE clause and the JOIN criteria. Predicates are a logical condition that evaluate to true, false, or unknown. They include things like IN or BETWEEN.
2. The optimizer then examines indexes on those columns.
3. The optimizer assesses the usefulness of each index by determining the selectivity of the clause (that is, how many rows will be returned) from statistics maintained on the index.
4. Constraints such as primary keys and foreign keys are also assessed and used by the optimizer to determine the selectivity of the objects in use in the query.
5. Finally, the optimizer estimates the least costly method of retrieving the qualifying rows, based on the information gathered in the previous steps.

Note Chapter [13](#) covers statistics in more depth.

To understand the significance of a WHERE clause column in a query, let's consider an example. Let's return to the original code listing that helped you understand what an index is; the query consisted of a SELECT statement without any WHERE clause, as follows:

```
SELECT p.ProductID,  
       p.Name,  
       p.StandardCost,  
       p.Weight  
FROM Production.Product p;
```

The query optimizer performs a clustered index scan, the equivalent of a table scan against a heap on a table that has a clustered index, to read the rows as shown in Figure [8-6](#) (switch on the Include Actual Execution Plan option by pressing Ctrl+M inside

a query window, as well as the Set Statistics IO option by right-clicking, selecting Query Options, and then selecting the appropriate check box on the Advanced tab).

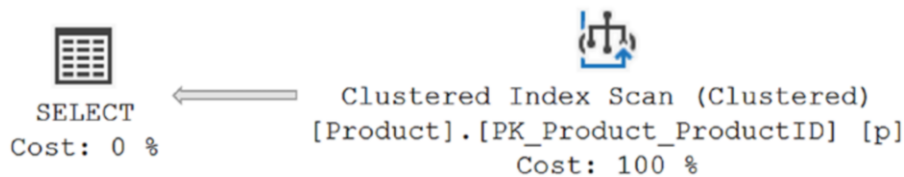


Figure 8-6. Execution plan with no WHERE clause

The number of logical reads reported by SET STATISTICS IO for the SELECT statement is as follows:

Table 'Product'. Scan count 1, logical reads 15

Note Capturing an execution plan can affect any time metrics you gather using almost any method. So, when measuring the time really counts, remember to turn off the execution plan capture.

To understand the effect of a WHERE clause column on the query optimizer’s decisions, let’s add a WHERE clause to retrieve a single row.

```
SELECT p.ProductID,
       p.Name,
       p.StandardCost,
       p.Weight
FROM Production.Product AS p
WHERE p.ProductID = 738;
```

With the WHERE clause in place, the query optimizer examines the WHERE clause column ProductID, identifies the availability of the index PK_Product_ProductId on column ProductId, assesses a high selectivity (that is, only one row will be returned) for the WHERE clause from the statistics on index PK_Product_ProductId, and decides to use that index to retrieve the data, as shown in Figure 8-7.

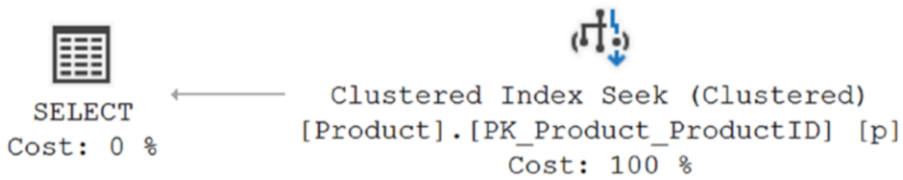


Figure 8-7. Execution plan with a WHERE clause

The resultant number of logical reads is as follows:

Table 'Product'. Scan count 0, logical reads 2

The behavior of the query optimizer shows that the WHERE clause column helps the optimizer choose an optimal indexing operation for a query. This is also applicable for a column used in the JOIN criteria between two tables. The optimizer looks for the indexes on the WHERE clause column or the JOIN criterion column and, if available, considers using the index to retrieve the rows from the table. The query optimizer considers indexes on the WHERE clause columns and the JOIN criteria columns while executing a query. Therefore, having indexes on the frequently used columns in the WHERE clause, the HAVING clause, and the JOIN criteria of a SQL query helps the optimizer avoid scanning a base table.

When the amount of data inside a table is so small that it fits onto a single page (8KB), a table scan may work better than an index seek. If you have a good index in place but you're still getting a scan, consider this effect.

Use Narrow Indexes

For best performance, you should use as narrow a data type as is practical when creating indexes. Narrow in this context means as small a data type as you realistically can. You should also avoid very wide data type columns in an index. Columns with string data types (CHAR, VARCHAR, NCHAR, and NVARCHAR) sometimes can be quite wide, as can binary and globally unique identifiers (GUIDs). Unless they are absolutely necessary, minimize the use of wide data type columns with large sizes in an index. You can create indexes on a combination of columns in a table. For the best performance, use as few columns in an index as necessary. However, use the columns you need to use to define a useful key for the index.

A narrow index can accommodate more rows in an 8KB index page than a wide index. This has the following effects:

- Reduces I/O (by having to read fewer 8KB pages)
- Makes database caching more effective because SQL Server can cache fewer index pages, consequently reducing the logical reads required for the index pages in the memory
- Reduces the storage space for the database

To understand how a narrow index can reduce the number of logical reads, create a test table with 20 rows and an index.

```
DROP TABLE IF EXISTS dbo.Test1;
GO
CREATE TABLE dbo.Test1 (C1 INT, C2 INT);

WITH Nums
      AS (SELECT 1 AS n
          UNION ALL
          SELECT n + 1
          FROM Nums
          WHERE n < 20
      )
INSERT INTO dbo.Test1
      (C1, C2)
      SELECT n,
            2
      FROM Nums;

CREATE INDEX iTest ON dbo.Test1(C1);
```

Since the indexed column is narrow (the INT data type is 4 bytes), all the index rows can be accommodated in one 8KB index page. As shown in Figure 8-8, you can confirm this in the dynamic management views associated with indexes. You may get an error if your database ID resolves to NULL.

```
SELECT i.name,
       i.type_desc,
       ddips.page_count,
```

```

        ddips.record_count,
        ddips.index_level
FROM sys.indexes i
    JOIN sys.dm_db_index_physical_stats( DB_ID(N'AdventureWorks2017'),
        OBJECT_ID(N'dbo.Test1'),
        NULL,
        NULL,
        'DETAILED'
        ) AS ddips
    ON i.index_id = ddips.index_id
WHERE i.object_id = OBJECT_ID(N'dbo.Test1');

```

	Name	type_desc	page_count	record_count	index_level
1	NULL	HEAP	1	20	0
2	iTest	NONCLUSTERED	1	20	0

Figure 8-8. Number of pages for a narrow, nonclustered index

The `sys.indexes` system table is stored in each database and contains the basic information on every index in the database. The dynamic management function `sys.dm_db_index_physical_stats` contains the more detailed information about the statistics on the index (you'll learn more about this DMV in Chapter 14). To understand the disadvantage of a wide index key, modify the data type of the indexed column `c1` from `INT` to `CHAR(500)` (`narrow_alter.sql` in the download).

```

DROP INDEX dbo.Test1.iTest;
ALTER TABLE dbo.Test1 ALTER COLUMN C1 CHAR(500);
CREATE INDEX iTest ON dbo.Test1(C1);

```

The width of a column with the `INT` data type is 4 bytes, and the width of a column with the `CHAR(500)` data type is 500 bytes. Because of the large width of the indexed column, two index pages are required to contain all 20 index rows. You can confirm this in the `sys.dm_db_index_physical_stats` dynamic management function by running the query against it again (see Figure 8-9).

	Name	type_desc	page_count	record_count	index_level
1	NULL	HEAP	2	25	0
2	iTest	NONCLUSTERED	2	20	0
3	iTest	NONCLUSTERED	1	2	1

Figure 8-9. Number of pages for a wide, nonclustered index

A large index key size increases the number of index pages, thereby increasing the amount of memory and disk activities required for the index. It is always recommended that the index key size be as narrow as you can make it.

Drop the test table before continuing.

```
DROP TABLE dbo.Test1;
```

Examine Column Uniqueness

Creating an index on columns with a very low range of possible unique values (such as `MaritalStatus`) will not benefit performance because the query optimizer will not be able to use the index to effectively narrow down the rows to be returned. Consider a `MaritalStatus` column with only two unique values: M and S. When you execute a query with the `MaritalStatus` column in the `WHERE` clause, you end up with a large number of rows from the table (assuming the distribution of M and S is relatively even), resulting in a costly table or clustered index scan. It is always preferable to have columns in the `WHERE` clause with lots of unique rows (or *high selectivity*) to limit the number of rows accessed. You should create an index on those columns to help the optimizer access a small result set.

Furthermore, while creating an index on multiple columns, which is also referred to as a *composite index*, column order matters. In many cases, using the most selective column first will help filter the index rows more efficiently.

Note The importance of column order in a composite index is explained later in the chapter in the “Consider Column Order” section.

From this, you can see that it is important to know the selectivity of a column before creating an index on it. You can find this by executing a query like this one; just substitute the table and column name:

```
SELECT COUNT(DISTINCT e.MaritalStatus) AS DistinctColValues,
       COUNT(e.MaritalStatus) AS NumberOfRows,
       (CAST(COUNT(DISTINCT e.MaritalStatus) AS DECIMAL)
        / CAST(COUNT(e.MaritalStatus) AS DECIMAL)) AS Selectivity,
       (1.0 / (COUNT(DISTINCT e.MaritalStatus))) AS Density
FROM HumanResources.Employee AS e;
```

Of course, you won't need to run this kind of query on every column or every index. This query is showing how some of the statistics that SQL Server uses are put together. You can see the statistics directly by using `DBCC SHOW_STATISTICS` or by querying the DMFs, `sys.dm_db_stats_histogram` and `sys.dm_db_stats_properties`. We'll cover all these in detail in Chapter 13.

The column with the highest number of unique values (or selectivity) can be the best candidate for indexing when referred to in a `WHERE` clause or a join criterion. You may also have the exceptional data where you have hundreds of rows of common data with only a few that are unique. The few will also benefit from an index. You can make this even more beneficial by using filtered indexes (discussed in more detail in Chapter 9).

To understand how the selectivity of an index key column affects the use of the index, take a look at the `MaritalStatus` column in the `HumanResources.Employee` table. If you run the previous query, you'll see that it contains only two distinct values in 290 rows, which is a selectivity of .0069 and a density of .5. A query to look for a `MaritalStatus` of M, as well as a particular `BirthDate` value, would look like this:

```
SELECT e.MaritalStatus,
       e.BirthDate
FROM HumanResources.Employee AS e
WHERE e.MaritalStatus = 'M'
      AND e.BirthDate = '1982-02-11';
```

This results in the execution plan in Figure 8-10 and the following I/O and elapsed time:

```
Table 'Employee'. Scan count 1, logical reads 9
CPU time = 0 ms, elapsed time = 2 ms.
```

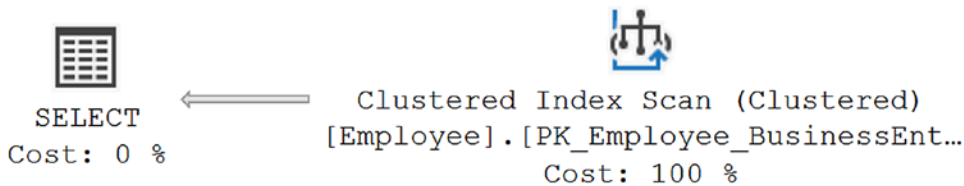


Figure 8-10. Execution plan with no index

The data is returned by scanning the clustered index (where the data is stored) to find the appropriate values where `MaritalStatus = 'M'`. If you were to place an index on the column, like so, and run the query again, the execution plan remains the same:

```
CREATE INDEX IX_Employee_Test ON HumanResources.Employee (MaritalStatus);
```

The data is just not selective enough for the index to be used, let alone be useful. If instead you use a composite index that looks like this:

```
CREATE INDEX IX_Employee_Test
ON HumanResources.Employee
(
    BirthDate,
    MaritalStatus
)
WITH (DROP_EXISTING = ON);
```

then, when you rerun the query, a completely different execution plan is generated. You can see it in Figure 8-11 along with the performance results.

Table 'Employee'. Scan count 1, logical reads 2
CPU time = 0 ms, elapsed time = 2 ms.

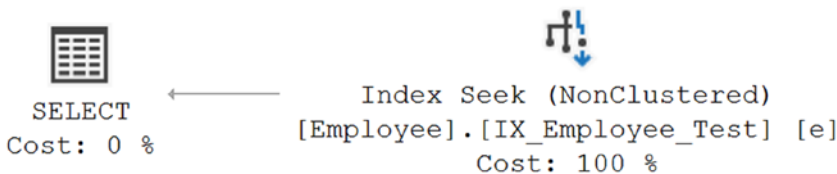


Figure 8-11. Execution plan with a composite index

Now you're doing better than you were with the clustered index scan. A nice clean Index Seek operation takes less than half the time to gather the data.

Although none of the columns in question would probably be selective enough on their own to make a decent index, except possibly the BirthDate column, together they provide enough selectivity for the optimizer to take advantage of the index offered.

It is possible to attempt to force the query to use the first test index you created. If you drop the compound index, create the original again, and then modify the query as follows by using a query hint to force the use of the original Index architecture:

```
CREATE INDEX IX_Employee_Test
ON HumanResources.Employee
(
    MaritalStatus
)
WITH (DROP_EXISTING = ON);

SELECT e.MaritalStatus,
       e.BirthDate
FROM HumanResources.Employee AS e WITH (INDEX(IX_Employee_Test))
WHERE e.MaritalStatus = 'M'
      AND e.BirthDate = '1982-02-11';
```

then the results and execution plan shown in Figure 8-12, while similar, are not the same.

Table 'Employee'. Scan count 1, logical reads 294
CPU time = 0 ms, elapsed time = 47 ms.

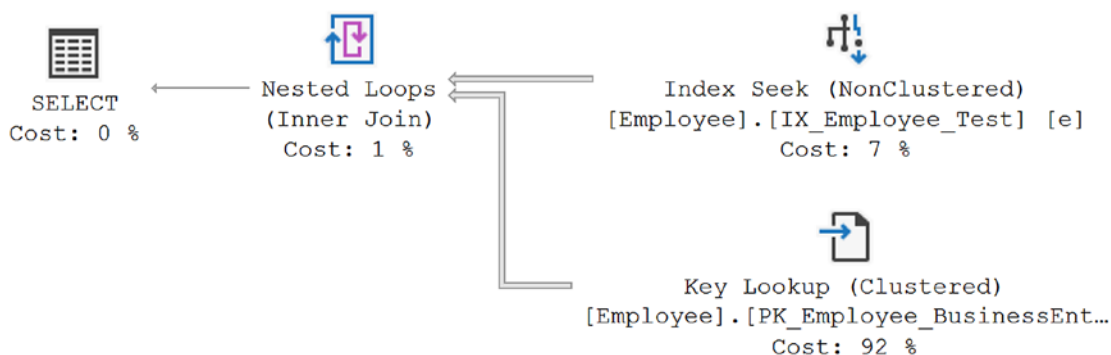


Figure 8-12. Execution plan when the index is chosen with a query hint

You see the same index seek, but the number of reads has increased radically, and the execution plan itself has changed. You now have a Nested Loops join and a Key Lookup operator added to the plan. Although forcing the optimizer to choose an index is possible, it clearly isn't always an optimal approach. A query hint takes away the optimizer's choices and forces it down paths that are frequently suboptimal. Hints don't consider changes in structure such as a new index that the optimizer could use to better effect. Hints also force the optimizer to ignore data changes that could result in better plans.

Note You will learn about key lookups in Chapter 12.

Another way to force a different behavior since SQL Server 2012 is the `FORCESEEK` query hint. `FORCESEEK` makes it so the optimizer will choose only Index Seek operations. If the query were rewritten like this:

```
SELECT e.MaritalStatus,
       e.BirthDate
FROM HumanResources.Employee AS e WITH (FORCESEEK)
WHERE e.MaritalStatus = 'M'
      AND e.BirthDate = '1982-02-11';
```

this query results in the same execution plan as Figure 8-12 and equally poor performance.

Limiting the options of the optimizer and forcing behaviors can in some situations help, but frequently, as shown with the results here, an increase in execution time and the number of reads is not helpful.

Before moving on, be sure to drop the test index from the table.

```
DROP INDEX HumanResources.Employee.IX_Employee_Test;
```

Examine the Column Data Type

The data type of an index matters. For example, an index search on integer keys is fast because of the small size and easy arithmetic manipulation of the `INTEGER` (or `INT`) data type. You can also use other variations of integer data types (`BIGINT`, `SMALLINT`, and `TINYINT`) for index columns, whereas string data types (`CHAR`, `VARCHAR`, `NCHAR`, and

NVARCHAR) require a string match operation, which is usually costlier than an integer match operation.

Suppose you want to create an index on one column and you have two candidate columns—one with an INTEGER data type and the other with a CHAR(4) data type. Even though the size of both data types is 4 bytes in SQL Server 2017 and Azure SQL Database, you should still prefer the INTEGER data type index. Look at arithmetic operations as an example. The value 1 in the CHAR(4) data type is actually stored as 1 followed by three spaces, a combination of the following four bytes: 0x35, 0x20, 0x20, and 0x20. The CPU doesn't understand how to perform arithmetic operations on this data, and therefore it converts to an integer data type before the arithmetic operations, whereas the value 1 in an integer data type is saved as 0x00000001. The CPU can easily perform arithmetic operations on this data.

Of course, most of the time, you won't have the simple choice between identically sized data types, allowing you to choose the more optimal type. Keep this information in mind when designing and building your indexes.

Consider Index Column Order

An index key is sorted on the first column of the index and then subsorted on the next column within each value of the previous column. The first column in a compound index is frequently referred to as the *leading edge* of the index. For example, consider Table 8-2.

Table 8-2. *Sample Table*

c1	c2
1	1
2	1
3	1
1	2
2	2
3	2

If a composite index is created on the columns (c1, c2), then the index will be ordered as shown in Table 8-3.

Table 8-3. *Composite Index on Columns (c1, c2)*

c1	c2
1	1
1	2
2	1
2	2
3	1
3	2

As shown in Table 8-3, the data is sorted on the first column (c1) in the composite index. Within each value of the first column, the data is further sorted on the second column (c2).

Therefore, the column order in a composite index is an important factor in the effectiveness of the index. You can see this by considering the following:

- Column uniqueness
- Column width
- Column data type

For example, suppose most of your queries on table t1 are similar to the following:

```
SELECT p.ProductID FROM Production.Product AS p
WHERE p.ProductSubcategoryID = 1;

SELECT p.ProductID FROM Production.Product AS p
WHERE p.ProductSubcategoryID = 1
AND p.ProductModelID = 19;
```

An index on (ProductSubcategoryID, ProductModelID) will benefit both the queries. But an index on (ProductModelID, ProductSubCategoryID) will not be helpful to both queries because it will sort the data initially on column ProductModelID, whereas the first SELECT statement needs the data to be sorted on column ProductSubCategoryID.

To understand the importance of column ordering in an index, consider the following example. In the `Person.Address` table, there is a column for `City` and another for `PostalCode`. Create an index on the table like this:

```
CREATE INDEX IX_Test ON Person.Address (City, PostalCode);
```

A simple `SELECT` statement run against the table that will use this new index will look something like this:

```
SELECT a.City,
       a.PostalCode
FROM Person.Address AS a
WHERE a.City = 'Dresden';
```

The I/O and execution time for the query is as follows:

```
Table 'Address'. Scan count 1, logical reads 2
CPU time = 0 ms, elapsed time = 0 ms. (or 164 microseconds in Extended
Events)
```

The execution plan in Figure 8-13 shows the use of the index.

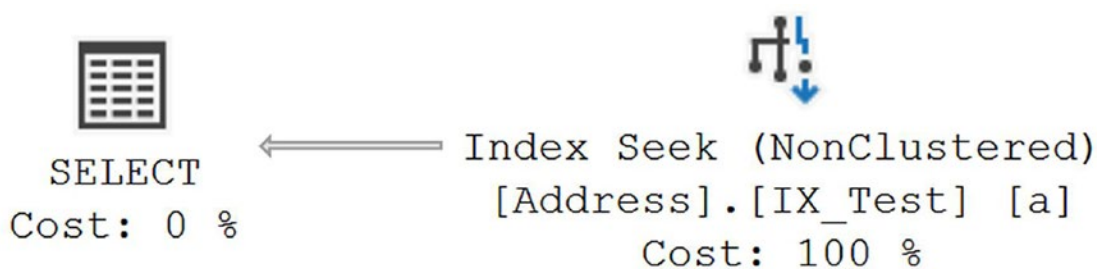


Figure 8-13. Execution plan for query against leading edge of index

So, this query is taking advantage of the leading edge of the index to perform a Seek operation to retrieve the data. If, instead of querying using the leading edge, you use another column in the index like the following query:

```
SELECT a.City,
       a.PostalCode
FROM Person.Address AS a
WHERE a.PostalCode = '01071';
```

The results are as follows:

Table 'Address'. Scan count 1, logical reads 108
CPU time = 0 ms, elapsed time = 2 ms.

The execution plan is clearly different, as you can see in Figure 8-14.



Figure 8-14. Execution plan for query against inner columns

Both queries return 31 rows from the same table, but the number of reads jumped from 2 to 108. You begin to see the difference between the Index Seek operation in Figure 8-13 and the Index Scan operation in Figure 8-14. The radical changes in I/O and time represents another advantage of a compound index, the covering index. This is covered in detail in Chapter 9.

When finished, drop the index.

```
DROP INDEX Person.Address.IX_Test;
```

Consider the Type of Index

In SQL Server, from all the different types of indexes available to you, most of the time you'll be working with the two main index types: *clustered* and *nonclustered*. Both types have a B-tree structure. The main difference between the two types is that the leaf pages in a clustered index are the data pages of the table and are therefore in the same order as the data to which they point. This means the clustered index is the table. As you proceed, you will see that the difference at the leaf level between the two index types becomes important when determining the type of index to use.

There are a number of other index types, and we'll cover them in more detail in Chapter 9.

Clustered Indexes

The leaf pages of a clustered index and the data pages of the table the index is on are one and the same. Because of this, table rows are physically sorted on the clustered index column, and since there can be only one physical order of the table data, a table can have only one clustered index.

Tip When you create a primary key constraint, SQL Server automatically creates it as a unique clustered index on the primary key if one does not already exist and if it is not explicitly specified that the index should be a unique nonclustered index. This is not a requirement; it's just default behavior. You can change the definition of the primary key prior to creating it on the table.

Heap Tables

As mentioned earlier in the chapter, a table with no clustered index is called a *heap table*. The data rows of a heap table are not stored in any particular order or linked to the adjacent pages in the table. This unorganized structure of the heap table usually increases the overhead of accessing a large heap table when compared to accessing a large nonheap table (a table with a clustered index).

Relationship with Nonclustered Indexes

There is an interesting relationship between a clustered index and the nonclustered indexes in SQL Server. An index row of a nonclustered index contains a pointer to the corresponding data row of the table. This pointer is called a *row locator*. The value of the row locator depends on whether the data pages are stored in a heap or on a clustered index. For a nonclustered index, the row locator is a pointer to the row identifier (RID) for the data row in a heap. For a table with a clustered index, the row locator is the clustered index key value.

For example, say you have a heap table with no clustered index, as shown in Table 8-4.

Table 8-4. *Data Page for a Sample Table*

RowID (Not a Real Column)	c1	c2	c3
1	A1	A2	A3
2	B1	B2	B3

A nonclustered index on column c1 in a heap will cause the row locator for the index rows to contain a pointer to the corresponding data row in the database table, as shown in Table 8-5.

Table 8-5. *Nonclustered Index Page with No Clustered Index*

c1	Row Locator
A1	Pointer to RID = 1
B1	Pointer to RID = 2

On creating a clustered index on column c2, the row locator values of the nonclustered index rows are changed. The new value of the row locator will contain the clustered index key value, as shown in Table 8-6.

Table 8-6. *Nonclustered Index Page with a Clustered Index on c2*

c1	Row Locator
A1	A2
B1	B2

To verify this dependency between a clustered and a nonclustered index, let's consider an example. In the AdventureWorks2017 database, the table `dbo.DatabaseLog` contains no clustered index, just a nonclustered primary key. If a query is run against it like the following, then the execution will look like Figure 8-15:


```

SELECT dl.DatabaseLogID,
       dl.PostTime
FROM dbo.DatabaseLog AS dl
WHERE dl.DatabaseLogID = 115;

```

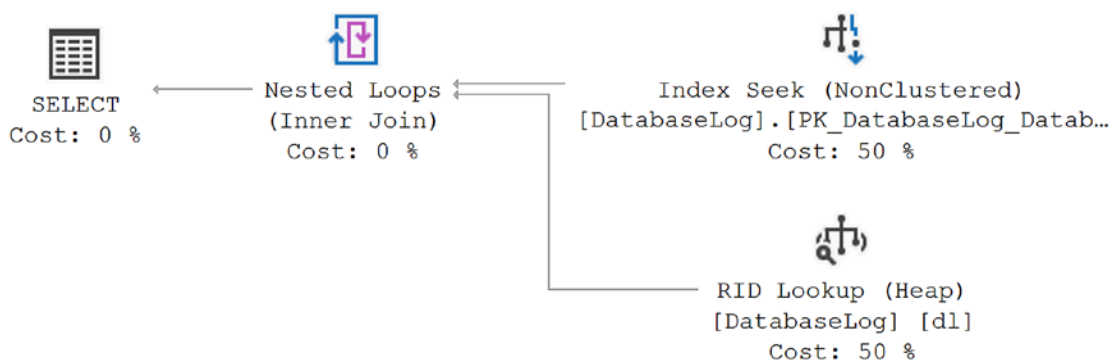


Figure 8-15. Execution plan against a heap

As you can see, the index was used in a Seek operation. But because the data is stored separately from the nonclustered index and that index doesn't contain all the columns needed to satisfy the query, an additional operation, the RID Lookup operation, is required to retrieve the data. The data from the two sources, the heap and the nonclustered index, are then joined through a Nested Loop operation. This is a classic example of what is known as a *lookup*, in this case an RID lookup, which is explained in more detail in the “Defining the Lookup” section. A similar query run against a table with a clustered index in place will look like this:

```

SELECT d.DepartmentID,
       d.ModifiedDate
FROM HumanResources.Department AS d
WHERE d.DepartmentID = 10;

```

Figure 8-16 shows this execution plan returned.

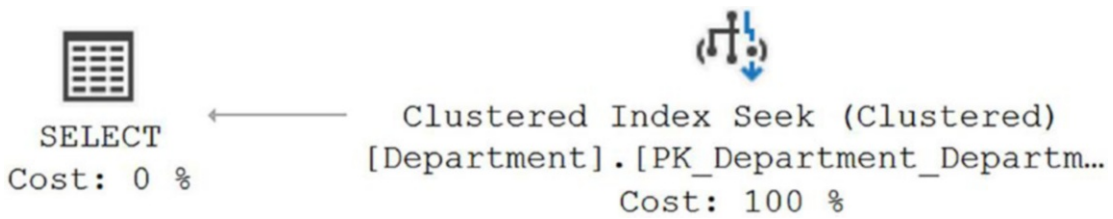


Figure 8-16. Execution plan with a clustered index

Although the primary key is used in the same way as the previous query, this time it's against a clustered index. This means the data is stored with the index, so the additional column doesn't require a lookup operation to get the data. Everything is returned by the simple Clustered Index Seek operation.

To navigate from a nonclustered index row to a data row, this relationship between the two index types requires an additional indirection for navigating the B-tree structure of the clustered index. Without the clustered index, the row locator of the nonclustered index would be able to navigate directly from the nonclustered index row to the data row in the base table. The presence of the clustered index causes the navigation from the nonclustered index row to the data row to go through the B-tree structure of the clustered index, since the new row locator value points to the clustered index key.

On the other hand, consider inserting an intermediate row in the clustered index key order or expanding the content of an intermediate row. For example, imagine a clustered index table containing four rows per page, with clustered index column values of 1, 2, 4, and 5. Adding a new row in the table with the clustered index value 3 will require space in the page between values 2 and 4. If enough space is not available in that position, a page split will occur on the data page (or clustered index leaf page). Even though the data page split will cause relocation of the data rows, the nonclustered index row locator values need not be updated. These row locators continue to point to the same logical key values of the clustered index key, even though the data rows have physically moved to a different location. In the case of a data page split, the row locators of the nonclustered indexes need not be updated. This is an important point since tables often have a large number of nonclustered indexes.

Things don't work the same way for heap tables. Page splits in a heap are not a common occurrence, and when heaps do split, they don't rearrange locations in the same way as clustered indexes. However, you can have rows move in a heap, usually because of updates causing the heap to not fit on its current page. Anything that causes the location

of rows to be moved in a heap results in a forwarding record being placed into the original location pointing to that new location, necessitating even more I/O activity.

Note Page splits and their effect on performance are explained in more detail in Chapter 14.

Clustered Index Recommendations

The relationship between a clustered index and a nonclustered index imposes some considerations on the clustered index, which are explained in the sections that follow.

Create the Clustered Index First

Since all nonclustered indexes hold clustered index keys within their index rows, the order of creation for nonclustered and clustered indexes is important. For example, if the nonclustered indexes are built before the clustered index is created, then the nonclustered index row locator will contain a pointer to the corresponding RID of the table. Creating the clustered index later will modify all the nonclustered indexes to contain clustered index keys as the new row locator value. This effectively rebuilds all the nonclustered indexes.

For the best performance, I recommend you create the clustered index *before* you create any nonclustered index. This allows the nonclustered indexes to have their row locator set to the clustered index keys at the time of creation. This does not have any effect on the final performance, but rebuilding the indexes may be quite a large job.

As part of creating the clustered index first, I also suggest you design the tables in your OLTP database around the clustered index. It should be the first index created because you should be storing your data as a clustered index by default.

For analysis and warehouse data, another option for data storage is available, the clustered columnstore index. We'll address that index in Chapter 9.

Keep Clustered Indexes Narrow

Since all nonclustered indexes hold the clustered keys as their row locator, for the best performance, keep the overall byte size of the clustered index as small as possible. If you create a wide clustered index, say `CHAR(500)`, in addition to having fewer rows per page in the cluster, this will add 500 bytes to every nonclustered index. Thus, keep the number

of columns in the clustered index to a minimum, and carefully consider the byte size of each column to be included in the clustered index. A column of the integer data type often makes a good candidate for a clustered index, whereas a string data type column will be a less-than-optimal choice. Conversely, choose the right key values for the clustered index, even if it means the key is wider. A wide key can hurt performance, but the wrong cluster key can hurt performance even more.

To understand the effect of a wide clustered index on a nonclustered index, consider this example. Create a small test table with a clustered index and a nonclustered index.

```
DROP TABLE IF EXISTS dbo.Test1;
GO
CREATE TABLE dbo.Test1 (
    C1 INT,
    C2 INT);

WITH Nums
AS (SELECT TOP (20)
    ROW_NUMBER() OVER (ORDER BY (SELECT 1)) AS n
    FROM master.sys.all_columns ac1
    CROSS JOIN master.sys.all_columns ac2)
INSERT INTO dbo.Test1 (
    C1,
    C2)
SELECT n,
    n + 1
FROM Nums;

CREATE CLUSTERED INDEX iClustered ON dbo.Test1 (C2);

CREATE NONCLUSTERED INDEX iNonClustered ON dbo.Test1 (C1);
```

Since the table has a clustered index, the row locator of the nonclustered index contains the clustered index key value. Therefore:

Width of the nonclustered index row = width of the nonclustered
index column + width of the clustered index column = size of INT
data type + size of INT data type

= 4 bytes + 4 bytes = 8 bytes

With this small size of a nonclustered index row, all the rows can be stored in one index page. You can confirm this by querying against the index statistics, as shown in Figure 8-17.

```
SELECT i.name,
       i.type_desc,
       s.page_count,
       s.record_count,
       s.index_level
FROM sys.indexes i
     JOIN sys.dm_db_index_physical_stats(DB_ID(N'AdventureWorks2017'),
                                           OBJECT_ID(N'dbo.Test1'),
                                           NULL,
                                           NULL,
                                           'DETAILED') AS s
      ON i.index_id = s.index_id
WHERE i.object_id = OBJECT_ID(N'dbo.Test1');
```

	name	type_desc	page_count	record_count	index_level
1	iClustered	CLUSTERED	1	20	0
2	iNonClustered	NONCLUSTERED	1	20	0

Figure 8-17. Number of index pages for a narrow index

To understand the effect of a wide clustered index on a nonclustered index, modify the data type of the clustered indexed column c2 from INT to CHAR(500).

```
DROP INDEX dbo.Test1.iClustered;
ALTER TABLE dbo.Test1 ALTER COLUMN C2 CHAR(500);
CREATE CLUSTERED INDEX iClustered ON dbo.Test1(C2);
```

Running the query against sys.dm_db_index_physical_stats again returns the result in Figure 8-18.

	name	type_desc	page_count	record_count	index_level
1	iClustered	CLUSTERED	2	20	0
2	iClustered	CLUSTERED	1	2	1
3	iNonClustered	NONCLUSTERED	2	20	0
4	iNonClustered	NONCLUSTERED	1	2	1

Figure 8-18. Number of index pages for a wide index

You can see that a wide clustered index increases the width of the nonclustered index row size. Because of the large width of the nonclustered index row, one 8KB index page can't accommodate all the index rows. Instead, two index pages will be required to store all 20 index rows. In the case of a large table, an expansion in the size of the nonclustered indexes because of a large clustered index key size can significantly increase the number of pages of the nonclustered indexes.

Therefore, a large clustered index key size not only affects its own width but also widens all nonclustered indexes on the table. This increases the number of index pages for all the indexes on the table, increasing the logical reads and disk I/Os required for the indexes.

Rebuild the Clustered Index in a Single Step

Because of the dependency of nonclustered indexes on the clustered index, rebuilding the clustered index as separate `DROP INDEX` and `CREATE INDEX` statements causes all the nonclustered indexes to be rebuilt twice. To avoid this, use the `DROP_EXISTING` clause of the `CREATE INDEX` statement to rebuild the clustered index in a single atomic step. Similarly, you can also use the `DROP_EXISTING` clause with a nonclustered index.

It's worth noting that in SQL Server 2005 and newer, when you perform a straight rebuild of a clustered index, you won't see the nonclustered indexes rebuilt as well.

Where Possible, Make the Clustered Index Unique

Because the clustered index is used to store the data, you must be able to find each row. While the clustered index doesn't have to be unique purely in terms of its definition and storage, if the key values are not unique, SQL Server would be unable to find the rows unless there was a way to make the cluster uniquely identify the location of each discrete row of data. So, SQL Server will add a value to a nonunique clustered index to make it unique. This value is called a *uniqueifier*. It adds to the size of your clustered index as well as all nonclustered indexes, as noted earlier. It also means a little bit of added

processing to get the unique value as each row gets inserted. For all these reasons, it makes sense to make the clustered index unique where you can. This is a big reason why the default behavior for primary keys is to make them a clustered index.

You don't *have* to make the clustered index unique. But you do need to take the uniquifier into account when you're defining your storage and indexes. Further, it's worth noting, since the uniquifier uses an integer data type, it limits the number of duplicate key values you can have to 2.1 billion duplicates for any one key (or keys) value. This shouldn't ever be a problem, but it is a possibility.

When to Use a Clustered Index

In certain situations, using a clustered index is helpful. I discuss these situations in the sections that follow.

Accessing the Data Directly

With all the data stored on the leaf pages of a clustered index, any time you access the cluster, the data is immediately available. One use for a clustered index is to support the most commonly used access path to the data. Any access of the clustered index does not require any additional reads to retrieve the data, which means seeks or scans against the clustered index do not require any additional reads to retrieve that data. This is another likely reason that Microsoft has made the primary key a clustered index by default. Since the primary key is frequently the most likely means of accessing data in a table, it serves well as a clustered index.

Just remember that the primary key being the clustered index is a default behavior but not necessarily the most common access path to the data. This could be through foreign key constraints, alternate keys in the table, or other columns. Plan and design the cluster with storage and access in mind, and you should be fine.

The clustered index works well as the primary path to the data only if you're accessing a considerable portion of the data within a table. If, on the other hand, you're accessing small subsets of the data, you might be better off with a nonclustered covering index. Also, you have to consider the number and types of columns that define the access path to the data. Since the key of a clustered index becomes the pointer for nonclustered indexes, excessively wide clustered keys can seriously impact performance and storage for nonclustered indexes.

As mentioned previously, if the majority of your queries are of the analysis variety with lots of aggregates, you may be better off storing the data with a clustered columnstore (covered in detail in Chapter 9).

Retrieving Presorted Data

Clustered indexes are particularly efficient when the data retrieval needs to be sorted (a covering nonclustered index is also useful for this). If you create a clustered index on the column or columns that you may need to sort by, then the rows will be physically stored in that order, eliminating the overhead of sorting the data after it is retrieved.

Let's see this in action. Create a test table as follows:

```
DROP TABLE IF EXISTS dbo.od;  
GO  
SELECT pod.*  
INTO dbo.od  
FROM Purchasing.PurchaseOrderDetail AS pod;
```

The new table `od` is created with data only. It doesn't have any indexes. You can verify the indexes on the table by executing the following, which returns nothing:

```
EXEC sp_helpindex 'dbo.od';
```

To understand the use of a clustered index, fetch a large range of rows ordered on a certain column.

```
SELECT od.*  
FROM dbo.od  
WHERE od.ProductID  
BETWEEN 500 AND 510  
ORDER BY od.ProductID;
```

You can obtain the cost of executing this query (without any indexes) from the STATISTICS IO output.

```
Table 'od'. Scan count 1, logical reads 78  
CPU time = 0 ms, elapsed time = 173 ms.
```


To improve the performance of this query, you should create an index on the WHERE clause column. This query requires both a range of rows and a sorted output. The result set requirement of this query meets the recommendations for a clustered index. Therefore, create a clustered index as follows and reexamine the cost of the query:

```
CREATE CLUSTERED INDEX i1 ON od(ProductID);
```

When you run the query again, the resultant cost of the query (with a clustered index) is as follows:

```
Table 'od'. Scan count 1, logical reads 8  
CPU time = 0 ms, elapsed time = 121 ms.
```

Creating the clustered index reduced the number of logical reads and therefore should contribute to the query performance improvement.

On the other hand, if you create a nonclustered index (instead of a clustered index) on the candidate column, then the query performance may be affected adversely. Let's verify the effect of a nonclustered index in this case.

```
DROP INDEX od.i1;  
CREATE NONCLUSTERED INDEX i1 on dbo.od(ProductID);
```

The resultant cost of the query (with a nonclustered index) is as follows:

```
Table 'od'. Scan count 1, logical reads 87  
CPU time = 0 ms, elapsed time = 163 ms.
```

The nonclustered index isn't even used directly in the resulting execution plan. Instead, you get a table scan, but the estimated costs for sorting the data in this new plan are different from the original table scan because of the added selectivity that the index provides the optimizer to estimate costs, even though the index isn't used. Drop the test table when you're done.

```
DROP TABLE dbo.od;
```

Poor Design Practices for a Clustered Index

In certain situations, you are better off not using a clustered index. I discuss these in the sections that follow.

Frequently Updatable Columns

If the clustered index columns are frequently updated, this will cause the row locator of all the nonclustered indexes to be updated accordingly, significantly increasing the cost of the relevant action queries. This also affects database concurrency by blocking all other queries referring to the same part of the table and the nonclustered indexes during that period. Therefore, avoid creating a clustered index on columns that are highly updatable.

Note Chapter 22 covers blocking in more depth.

To understand how the cost of an UPDATE statement that modifies only a clustered key column is increased by the presence of nonclustered indexes on the table, consider the following example. The Sales.SpecialOfferProduct table has a composite clustered index on the primary key, which is also the foreign key from two different tables; this is a classic many-to-many join. In this example, I update one of the two columns using the following statement (note the use of the transaction to keep the test data intact):

```
BEGIN TRAN;
SET STATISTICS IO ON;
UPDATE Sales.SpecialOfferProduct
SET ProductID = 345
WHERE SpecialOfferID = 1
    AND ProductID = 720;
SET STATISTICS IO OFF;
ROLLBACK TRAN;
```

The STATISTICS IO output shows the reads necessary.

```
Table 'Product'. Scan count 0, logical reads 2
Table 'SalesOrderDetail'. Scan count 1, logical reads 1248
Table 'SpecialOfferProduct'. Scan count 0, logical reads 10
```

If you added a nonclustered index to the table, you would see the reads increase, as shown here:

```
CREATE NONCLUSTERED INDEX ixTest
ON Sales.SpecialOfferProduct (ModifiedDate);
```

When you run the same query again, the output of `STATISTICS IO` changes for the `SpecialOfferProduct` table.

Table 'Product'. Scan count 0, logical reads 2

Table 'SalesOrderDetail'. Scan count 1, logical reads 1248

Table 'SpecialOfferProduct'. Scan count 0, logical reads 19

The number of reads caused by the update of the clustered index is increased with the addition of the nonclustered index. Be sure to drop the index.

```
DROP INDEX Sales.SpecialOfferProduct.ixTest;
```

Wide Keys

Since all nonclustered indexes hold the clustered keys as their row locator, for performance reasons you should avoid creating a clustered index on a very wide column (or columns) or on too many columns. As explained in the preceding section, a clustered index must be as narrow as is practical.

Nonclustered Indexes

A nonclustered index does not affect the order of the data in the table pages because the leaf pages of a nonclustered index and the data pages of the table are separate. A pointer (the row locator) is required to navigate from an index row in the nonclustered index to the data row, whether stored on a cluster or in a heap. As you learned in the earlier “Clustered Indexes” section, the structure of the row locator depends on whether the data pages are stored in a heap or a clustered index. For a heap, the row locator is a pointer to the RID for the data row; for a table with a clustered index, the row locator is the clustered index key; for a table with a clustered columnstore, the row locator is an 8-byte value consisting of the columnstore’s `row_group_id` and `tuple_id`.

Nonclustered Index Maintenance

The row locator value of the nonclustered indexes continues to have the same clustered index value, even when the clustered index rows are physically relocated.

In a table that is a heap, where there is no clustered index, to optimize this maintenance cost, SQL Server adds a pointer to the old data page to point to the new data page after a page split, instead of updating the row locator of all the relevant

nonclustered indexes. Although this reduces the maintenance cost of the nonclustered indexes, it increases the navigation cost from the nonclustered index row to the data row within the heap since an extra link is added between the old data page and the new data page. Therefore, having a clustered index as the row locator decreases this overhead associated with the nonclustered index.

When a table is a clustered columnstore index, the storage values of exactly what is stored where changes as the index is rebuilt and data moves from the delta store into compressed storage. This would lead to all sorts of issues except a new bit of functionality within the clustered columnstore index allows for a mapping between where the nonclustered index thought the value was and where it actually is. Funny enough, this is called the *mapping index*. Values are added to it as the locations of data change within the clustered columnstore. It can slightly slow nonclustered index usage when the table data is contained in a clustered columnstore.

Defining the Lookup Operation

When a query requests columns that are not part of the nonclustered index chosen by the optimizer, a lookup is required. This may be a key lookup when going against a clustered index, columnstore or not, or an RID lookup when performed against a heap. In the past, the common term for these lookups came from the old definition name, *bookmark lookup*. That term is being used less and less since people haven't seen that phrase in execution plans since SQL Server 2000. Now you just refer to it as a lookup and then define the type, key, or RID. The lookup fetches the corresponding data row from the table by following the row locator value from the index row, requiring a logical read on the data page besides the logical read on the index page and a join operation to put the data together in a common output. However, if all the columns required by the query are available in the index itself, then access to the data page is not required. This is known as a *covering index*.

These lookups are the reason that large result sets are better served with a clustered index. A clustered index doesn't require a lookup since the leaf pages and data pages for a clustered index are the same.

Note Chapter 12 covers lookup operations in more detail.

Nonclustered Index Recommendations

Since a table can have only one clustered index, you can use the flexibility of multiple nonclustered indexes to help improve performance. I explain the factors that decide the use of a nonclustered index in the following sections.

When to Use a Nonclustered Index

A nonclustered index is most useful when all you want to do is retrieve a small number of rows and columns from a large table. As the number of columns to be retrieved increases, the ability to have a covering index decreases. Then, if you're also retrieving a large number of rows, the overhead cost of any lookup rises proportionately. To retrieve a small number of rows from a table, the indexed column should have a high selectivity.

Furthermore, there will be indexing requirements that won't be suitable for a clustered index, as explained in the "Clustered Indexes" section.

- Frequently updatable columns
- Wide keys

In these cases, you can use a nonclustered index since, unlike a clustered index, it doesn't affect other indexes in the table. A nonclustered index on a frequently updatable column isn't as costly as having a clustered index on that column. The UPDATE operation on a nonclustered index is limited to the base table and the nonclustered index. It doesn't affect any other nonclustered indexes on the table. Similarly, a nonclustered index on a wide column (or set of columns) doesn't increase the size of any other index, unlike that with a clustered index. However, remain cautious, even while creating a nonclustered index on a highly updatable column or a wide column (or set of columns) since this can increase the cost of action queries, as explained earlier in the chapter.

Tip A nonclustered index can also help resolve blocking and deadlock issues. I cover this in more depth in Chapters [21](#) and [22](#).

When Not to Use a Nonclustered Index

Nonclustered indexes are not suitable for queries that retrieve a large number of rows, unless they're covering indexes. Such queries are better served with a clustered index, which doesn't require a separate lookup to retrieve a data row. Since a lookup requires additional logical reads to get to the data page besides the logical read on the nonclustered index page, the cost of a query using a nonclustered index increases significantly for a large number of rows, such as when in a loop join that requires one lookup after another. The SQL Server query optimizer takes this cost into effect and accordingly can discard the nonclustered index when retrieving a large result set. Nonclustered indexes are also not as useful as columnstore indexes for analytics-style queries with more aggregates.

If your requirement is to retrieve a large result set from a table, then having a nonclustered index on the filter criterion (or the join criterion) column will probably not be useful unless you use a special type of nonclustered index called a *covering index*. I describe this index type in detail in [Chapter 9](#).

Clustered vs. Nonclustered Indexes

The main considerations in choosing between a clustered and a nonclustered index are as follows:

- Number of rows to be retrieved
- Data-ordering requirement
- Index key width
- Column update frequency
- Lookup cost
- Any disk hot spots

Benefits of a Clustered Index over a Nonclustered Index

When deciding upon a type of index on a table with no indexes, the clustered index is usually the preferred choice. Because the index page and the data pages are the same, the clustered index doesn't have to jump from the index row to the base row as is required in the case of a noncovering nonclustered index.

To understand how a clustered index can outperform a nonclustered index in these circumstances, even in retrieving a small number of rows, create a test table with a high selectivity for one column.

```
DROP TABLE IF EXISTS dbo.Test1;
CREATE TABLE dbo.Test1 (
    C1 INT,
    C2 INT);

WITH Nums
AS (SELECT TOP (10000)
     ROW_NUMBER() OVER (ORDER BY (SELECT 1)) AS n
    FROM master.sys.all_columns AS ac1
    CROSS JOIN master.sys.all_columns AS ac2)
INSERT INTO dbo.Test1 (
    C1,
    C2)
SELECT n,
       2
FROM Nums;
```

The following SELECT statement fetches only 1 out of 10,000 rows from the table:

```
SELECT t.C1,
       t.C2
FROM dbo.Test1 AS t
WHERE C1 = 1000;
```

This query results in the graphical execution plan shown in Figure 8-19 and the output of SET STATISTICS IO and STATISTICS TIME as follows:

```
Table 'Test1'. Scan count 1, logical reads 22
CPU time = 0 ms, elapsed time = 0 ms.
```

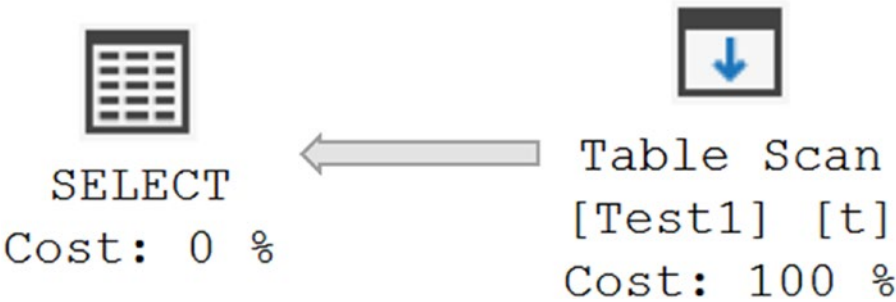


Figure 8-19. Execution plan with no index

Considering the small size of the result set retrieved by the preceding SELECT statement, a nonclustered column on c1 can be a good choice.

```
CREATE NONCLUSTERED INDEX incl ON dbo.Test1(C1);
```

You can run the same SELECT command again. Since retrieving a small number of rows through a nonclustered index is more economical than a table scan, the optimizer used the nonclustered index on column c1, as shown in Figure 8-20. The number of logical reads reported by STATISTICS IO is as follows:

```
Table 'Test1'. Scan count 1, logical reads 3  
CPU time = 0 ms, elapsed time = 0 ms.
```

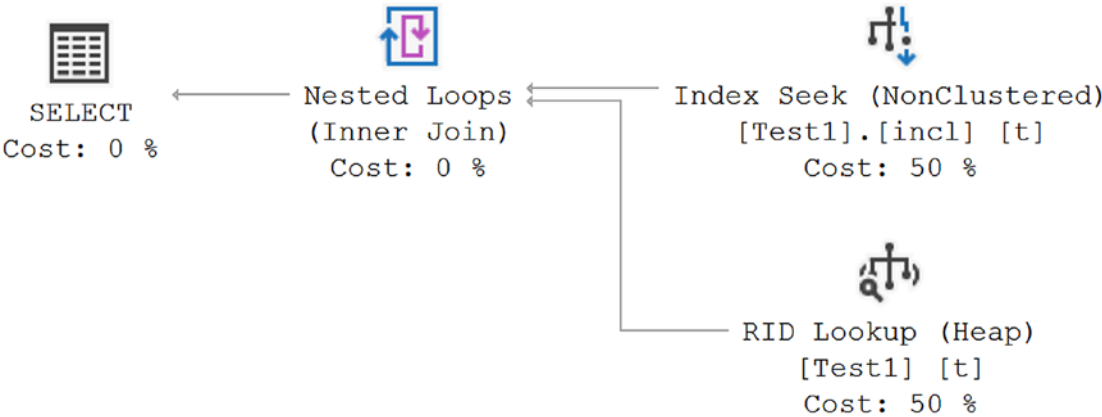


Figure 8-20. Execution plan with a nonclustered index

Even though retrieving a small result set using a column with high selectivity is a good pointer toward creating a nonclustered index on the column, a clustered index on the same column can be equally beneficial or even better. To evaluate how the clustered

index can be more beneficial than the nonclustered index, create a clustered index on the same column.

```
CREATE CLUSTERED INDEX icl ON dbo.Test1(C1);
```

Run the same SELECT command again. From the resultant execution plan (shown later in Figure 8-22) of the preceding SELECT statement, you can see that the optimizer used the clustered index (instead of the nonclustered index) even for a small result set. The number of logical reads for the SELECT statement decreased from three to two (Figure 8-21). You get this change in behavior because the clustered index is inherently a covering index, containing all the columns of the table.

Table 't1'. Scan count 1, logical reads 2

CPU time = 0 ms, elapsed time = 0 ms.

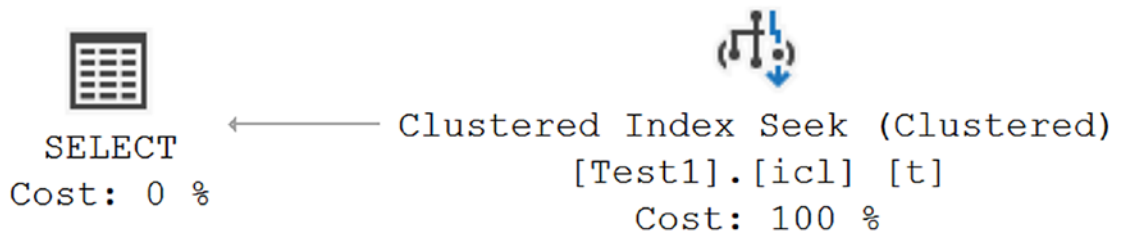


Figure 8-21. Execution plan with a clustered index

Note Because a table can have only one clustered index and that index is where the data is stored, I would generally reserve the clustered index for the most frequently used access path to the data.

Benefits of a Nonclustered Index over a Clustered Index

As you learned in the previous section, a nonclustered index is preferred over a clustered index in the following situations:

- When the index key size is large.
- To help avoid blocking by having a database reader work on the pages of a nonclustered index, while a database writer modifies other columns (not included in the nonclustered index) in the data page;

in this case, the writer working on the data page won't block a reader that can get all the required column values from the nonclustered index without hitting the base table. I'll explain this in detail in Chapter 13.

- When all the columns (from a table) referred to by a query can be safely accommodated in the nonclustered index itself, as explained in this section.
- When you're doing a point or limited range query against a clustered columnstore index. The clustered columnstore index supports analytical style queries very well, but it doesn't do point lookups well at all. That's why you add a nonclustered index just for the point lookup.

As already established, the data-retrieval performance when using a nonclustered index is generally poorer than that when using a clustered index because of the cost associated with jumping from the nonclustered index rows to the data rows in the base table. In cases where the jump to the data rows is not required, the performance of a nonclustered index should be just as good as—or even better than—a clustered index. This is possible if the nonclustered index, the key plus any included columns at the page level, includes all the columns required from the table.

To understand the situation in which a nonclustered index can outperform a clustered index, consider the following example. Assume for these purposes that you need to examine the credit cards that are expiring between the months of June 2008 and September 2008. You may have a query that returns a large number of rows and looks like this:

```
SELECT cc.CreditCardID,
       cc.CardNumber,
       cc.ExpMonth,
       cc.ExpYear
FROM Sales.CreditCard cc
WHERE cc.ExpMonth
      BETWEEN 6 AND 9
      AND cc.ExpYear = 2008
ORDER BY cc.ExpMonth;
```

The following are the I/O and time results. Figure 8-22 shows the execution plan.

Table 'CreditCard'. Scan count 1, logical reads 189

CPU time = 0 ms, elapsed time = 176 ms.

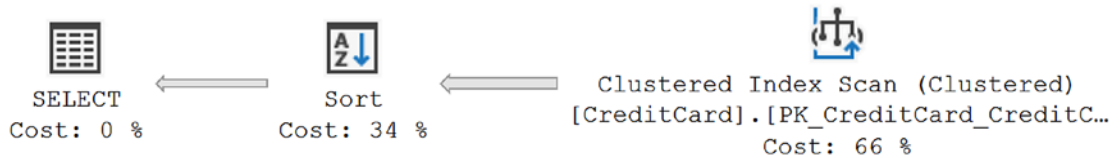


Figure 8-22. Execution plan scanning the clustered index

The clustered index is on the primary key, and although most access against the table may be through that key, making the index useful, the clustered index in this instance is just not performing in the way you need. Although you could expand the definition of the index to include all the other columns in the query, they're not really needed to make the clustered index function, and they would interfere with the operation of the primary key. Instead, you can use the INCLUDE operation to store the columns defined within it at the leaf level of the index. They don't affect the key structure of the index in any way but provide the ability, through the sacrifice of some additional disk space, to make a nonclustered index covering (covered in more detail later). In this instance, creating a different index is in order.

```
CREATE NONCLUSTERED INDEX ixTest
ON Sales.CreditCard
(
    ExpYear,
    ExpMonth)
INCLUDE
(
    CardNumber);
```

Now when the query is run again, this is the result:

Table 'CreditCard'. Scan count 1, logical reads 12

CPU time = 0 ms, elapsed time = 152 ms.

Figure 8-23 shows the corresponding execution plan.