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.
```

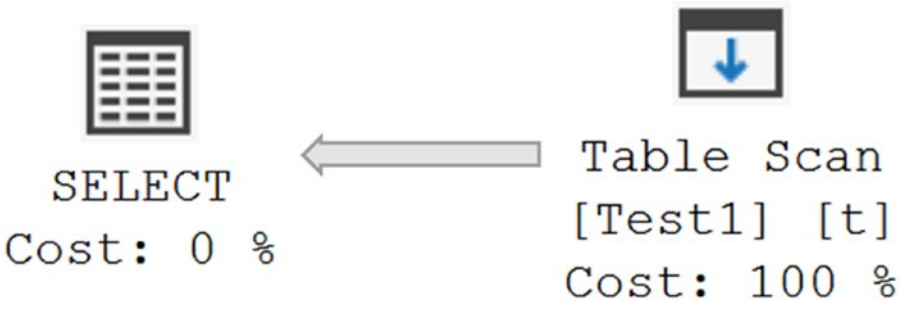

Figure 8-19. *Execution plan with no index*

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.
```

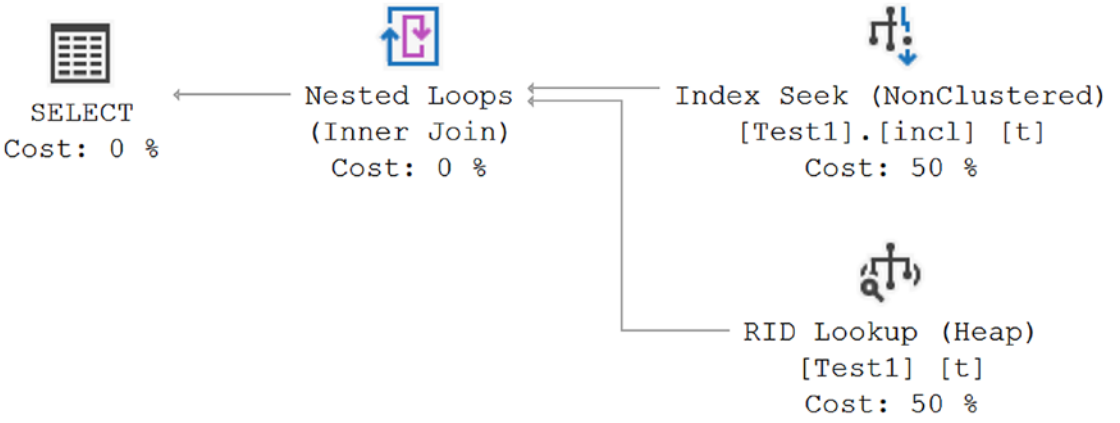

Figure 8-20. *Execution plan with a nonclustered index*

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.
```

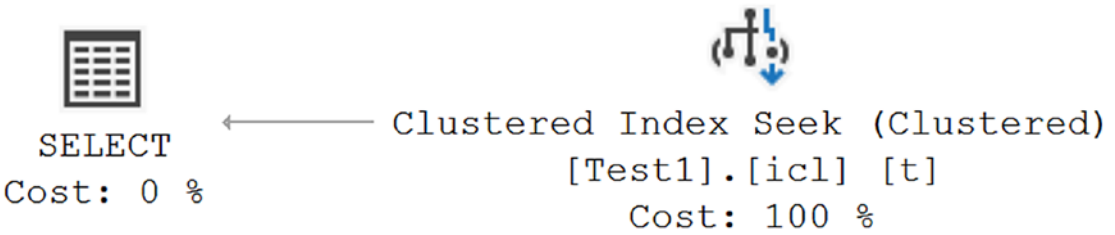

Figure 8-21. *Execution plan with a clustered index*

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.
```

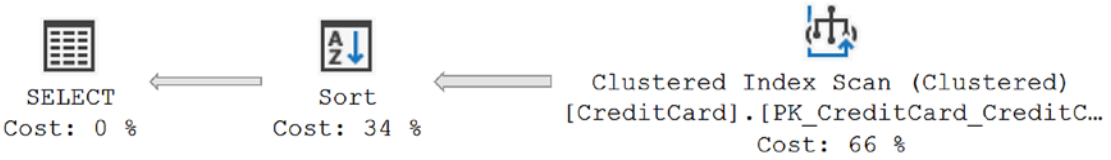

Figure 8-22. *Execution plan scanning the clustered index*

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.

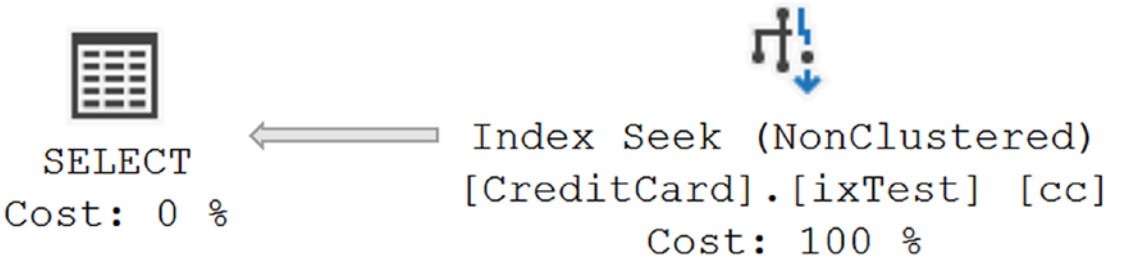

Figure 8-23. *Execution plan with a nonclustered index*

In this case, the `SELECT` statement doesn't include any column that requires a jump from the nonclustered index page to the data page of the table, which is what usually makes a nonclustered index costlier than a clustered index for a large result set and/or sorted output. This kind of nonclustered index is called a *covering index*.

It's also worth noting that I experimented with which column to put into the leading edge of the index, `ExpMonth` or `ExpYear`. After testing each, the reads associated with having `ExpMonth` first were 37 as compared to the 12 with `ExpYear`. That results from having to look through fewer pages with the year filtering first rather than the month. Remember to validate your choices when creating indexes with thorough testing.

Clean up the index after the testing is done.

```
DROP INDEX Sales.CreditCard.ixTest;
```

Summary

In this chapter, you learned that indexing is an effective method for reducing the number of logical reads and disk I/O for a query. Although an index may add overhead to action queries, even action queries such as `UPDATE` and `DELETE` can benefit from an index.

To decide the index key columns for a particular query, evaluate the `WHERE` clause and the join criteria of the query. Factors such as column selectivity, width, data type, and column order are important in deciding the columns in an index key. Since an index is mainly useful in retrieving a small number of rows, the selectivity of an indexed column should be very high. It is important to note that nonclustered indexes contain the value of a clustered index key as their row locator because this behavior greatly influences the selection of an index type.

In the next chapter, you will learn more about other functionality and other types of indexes available to help you tune your queries.

CHAPTER 9

Index Analysis

In the previous chapter I introduced the concepts surrounding B-tree indexes. This chapter takes that information and adds more functionality and more indexes. There's a lot of interesting interaction between indexes that you can take advantage of. There are also a number of settings that affect the behavior of indexes that I didn't address in the preceding chapter. I'll show you methods to squeeze even more performance out of your system. Most importantly, we dig into the details of the columnstore indexes and the radical improvement in performance that they can provide for analytical queries.

In this chapter, I cover the following topics:

- Advanced indexing techniques
- Columnstore indexes
- Special index types
- Additional characteristics of indexes

Advanced Indexing Techniques

Here are a few of the more advanced indexing techniques that you can consider:

- *Covering indexes*: These were introduced in Chapter 8.
- *Index intersections*: Use multiple nonclustered indexes to satisfy all the column requirements (from a table) for a query.
- *Index joins*: Use the index intersection and covering index techniques to avoid hitting the base table.
- *Filtered indexes*: To be able to index fields with odd data distributions or sparse columns, you can apply a filter to an index so that it indexes only some data.

G. Fritchey, *SQL Server 2017 Query Performance Tuning*, https://doi.org/10.1007/978-1-4842-3888-2_9

- *Indexed views*: These materialize the output of a view on disk.
- *Index compression*: The storage of indexes can be compressed through SQL Server, putting more rows of data on a page and improving performance.

I cover these topics in more detail in the following sections.

Covering Indexes

A *covering index* is a nonclustered index built upon all the columns required to satisfy a SQL query without going to the heap or the clustered index. If a query encounters an index and does not need to refer to the underlying structures at all, then the index can be considered a covering index.

For example, in the following `SELECT` statement, irrespective of where the columns are used within the statement, all the columns (`StateProvinceId` and `PostalCode`) should be included in the nonclustered index to cover the query fully:

```
SELECT  a.PostalCode
FROM    Person.Address AS a
WHERE   a.StateProvinceID = 42;
```

Then all the required data for the query can be obtained from the nonclustered index page, without accessing the data page. This helps SQL Server save logical and physical reads. If you run the query, you'll get the following I/O and execution time as well as the execution plan in Figure 9-1:

```
Table 'Address'. Scan count 1, logical reads 19
CPU time = 0 ms,  elapsed time = 0 ms.
```

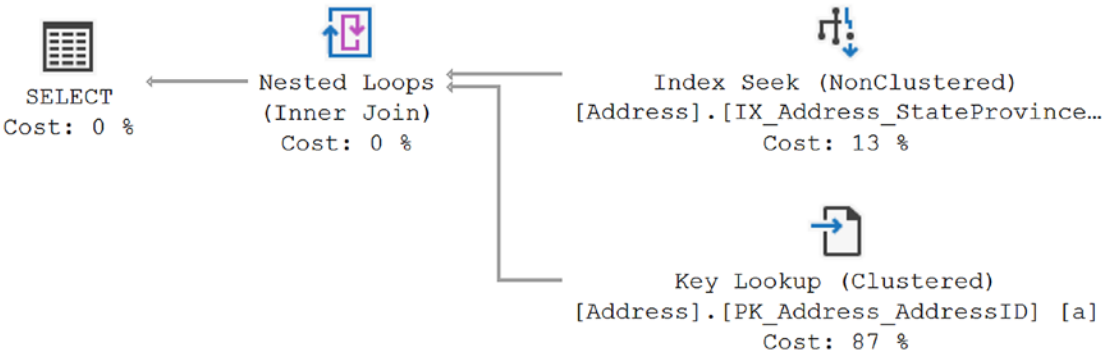

Figure 9-1. *Query without a covering index*

Here you have a classic lookup with the `Key Lookup` operator pulling the `PostalCode` data from the clustered index and joining it with the `Index Seek` operator against the `IX_Address_StateProvinceId` index.

Although you can re-create the index with both key columns, another way to make an index a covering index is to use the new `INCLUDE` operator. This stores data with the index without changing the structure of the index. I'll cover the details of why to use the `INCLUDE` operator a little later. Use the following to re-create the index:

```
CREATE NONCLUSTERED INDEX IX_Address_StateProvinceID
ON Person.Address
(
    StateProvinceID ASC
)
INCLUDE
(
    PostalCode
)
WITH (DROP_EXISTING = ON);
```

If you rerun the query, the execution plan (Figure 9-2), I/O, and execution time change. (Also, it's worth noting, 0ms is not the correct execution time. Using an Extended Events session, which records execution time in microseconds (µs), it's 177µs.

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

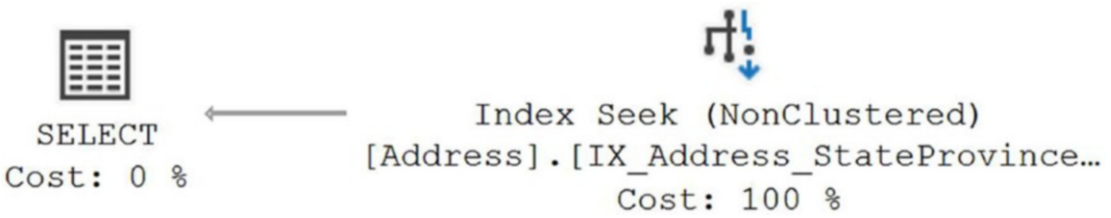

Figure 9-2. *Query with a covering index*

The reads have dropped from 19 to 2, and the execution plan is just about as simple as possible; it's a single Index Seek operation against the new and improved index, which is now covering. A covering index is a useful technique for reducing the number of logical reads of a query. Adding columns using the INCLUDE statement makes this functionality easier to achieve without adding to the number of columns in an index or the size of the index key since the included columns are stored only at the leaf level of the index.

The INCLUDE is best used in the following cases:

- You don't want to increase the size of the index keys, but you still want to make the index a covering index.
- You have a data type that cannot be an index key column but can be added to the nonclustered index through the INCLUDE command.
- You've already exceeded the maximum number of key columns for an index (although this is a problem best avoided).

Before continuing, put the index back into its original format.

```
CREATE NONCLUSTERED INDEX IX_Address_StateProvinceID
ON Person.Address
(
    StateProvinceID ASC
)
WITH (DROP_EXISTING = ON);
```

A Pseudoclustered Index

The covering index physically organizes the data of all the indexed columns in a sequential order. Thus, from a disk I/O perspective, a covering index that doesn't use included columns becomes a clustered index for all queries satisfied completely by the columns in the covering index. If the result set of a query requires a sorted output, then the covering index can be used to physically maintain the column data in the same order as required by the result set—it can then be used in the same way as a clustered index for sorted output. As shown in the previous example, covering indexes can give better performance than clustered indexes for queries requesting a range of rows and/or sorted output. The included columns are not part of the key and therefore wouldn't offer the same benefits for ordering as the key columns of the index.