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 I0 option by right-clicking, selecting Query Options, and then selecting the appropriate check box on the Advanced tab).

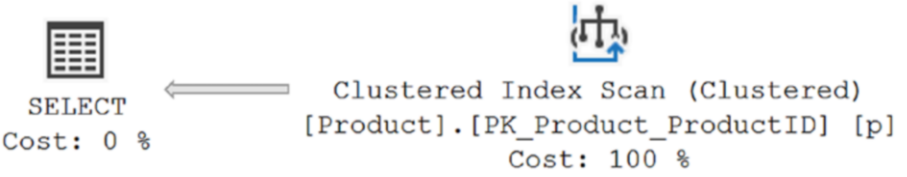

Figure 8-6. *Execution plan with no WHERE clause*

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 `Productld`, assesses a high selectivity (that is, only one row will be returned) for the `WHERE` clause from the statistics on index `PK_Product_Productld`, and decides to use that index to retrieve the data, as shown in Figure 8-7.

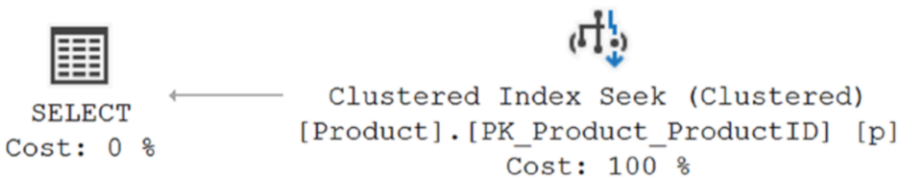

Figure 8-7. *Execution plan with a WHERE clause*

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.
```

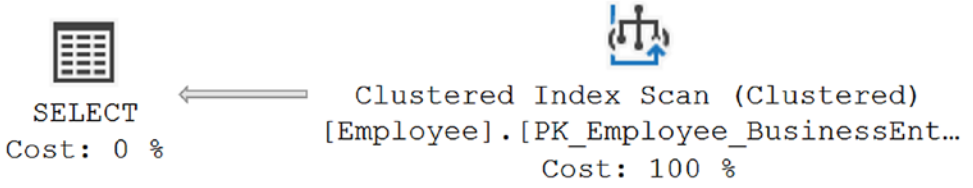

Figure 8-10. *Execution plan with no index*

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.
```

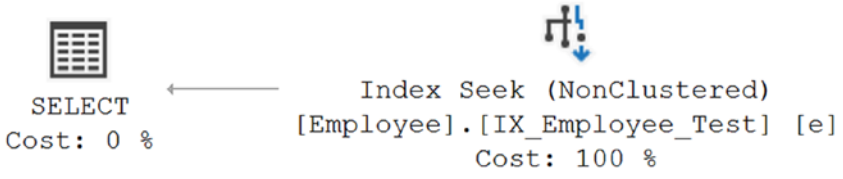

Figure 8-11. *Execution plan with a composite index*

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.
```

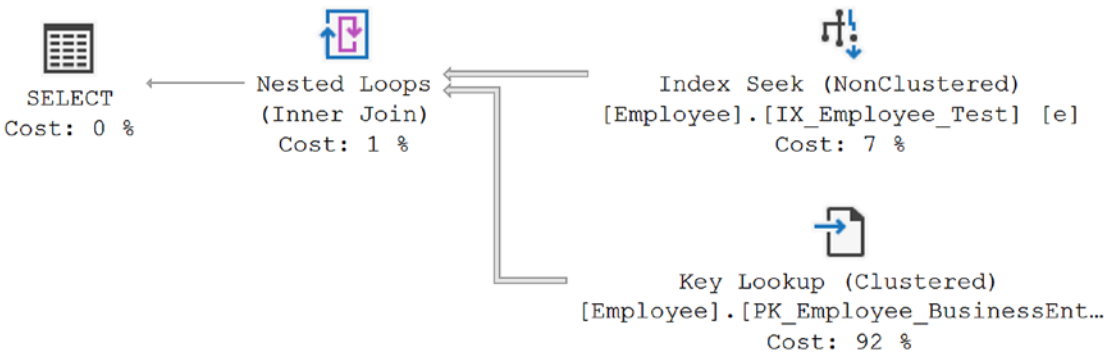

Figure 8-12. *Execution plan when the index is chosen with a query hint*

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.

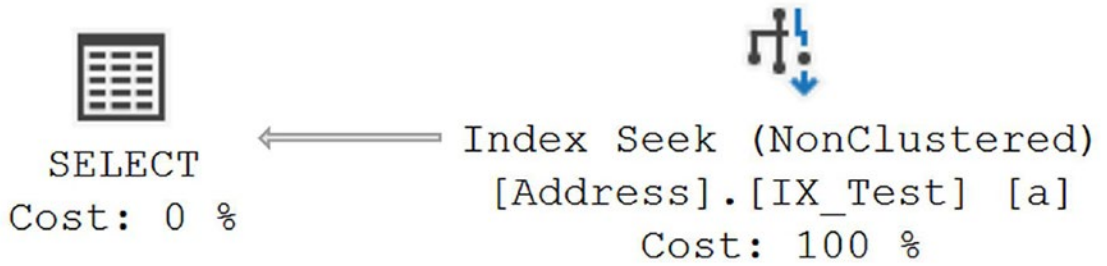

Figure 8-13. *Execution plan for query against leading edge of index*

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.

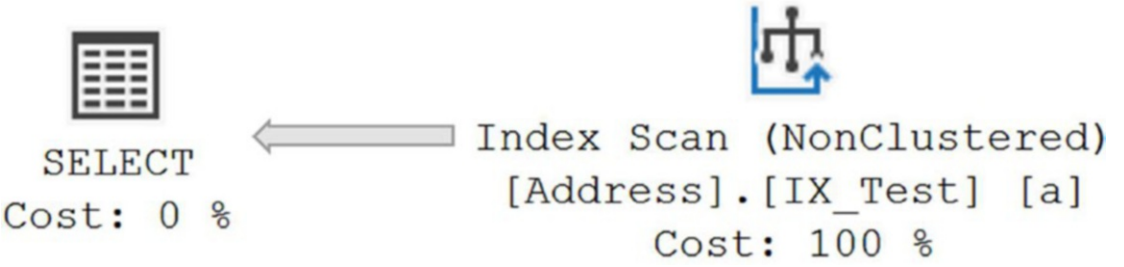

Figure 8-14. *Execution plan for query against inner columns*

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;
```

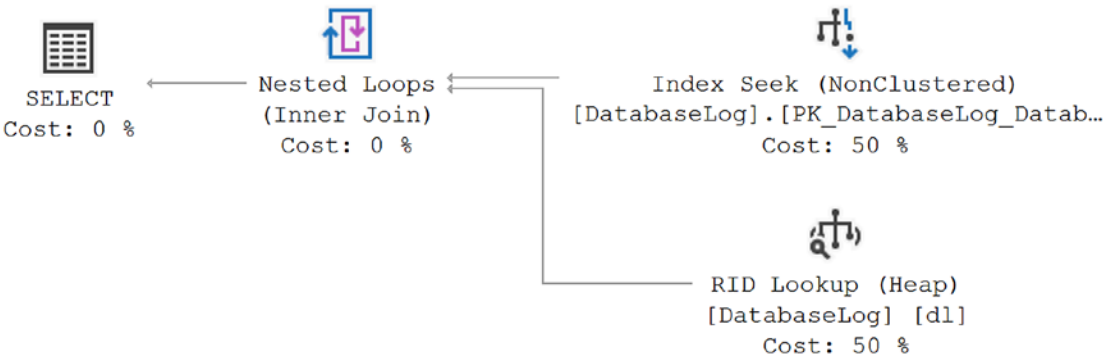

Figure 8-15. *Execution plan against a heap*

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.

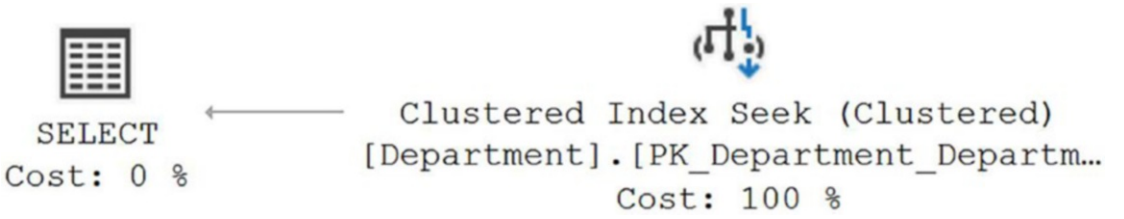

Figure 8-16. *Execution plan with a clustered index*

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
```