indicating the missing statistics information on the data access operators, the clustered index scans. If you modify the query to reference table `Test2` as the first table in the `FROM` clause, then the optimizer selects table `Test2` as the outer table of the nested loop join operation. Figure 13-14 shows the execution plan.

```
SELECT  Test1.Test1_C2,
        Test2.Test2_C2
FROM    dbo.Test2
JOIN    dbo.Test1
        ON Test1.Test1_C2 = Test2.Test2_C2
WHERE   Test1.Test1_C2 = 2;
```

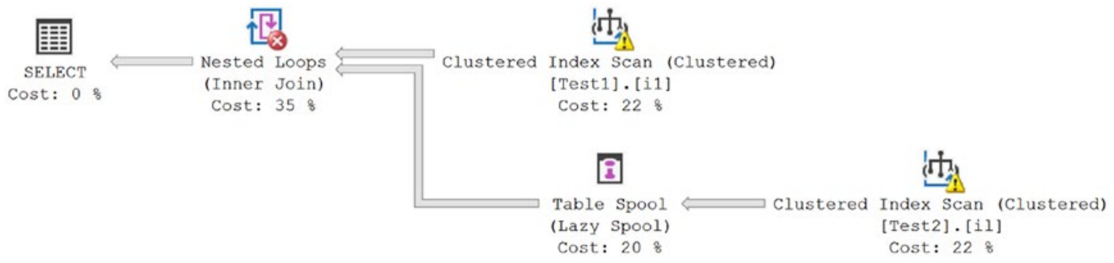


***Figure 13-14.** Execution plan with AUTO_CREATE_STATISTICS OFF (a variation)*

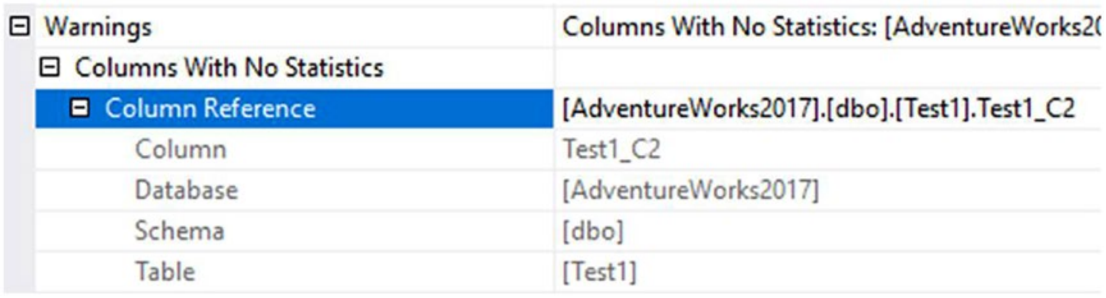
You can see that turning off the auto create statistics feature has a negative effect on performance by comparing the cost of this query with and without statistics on a nonindexed column. Table 13-3 shows the difference in the cost of this query.

***Table 13-3.** Cost Comparison of a Query with and Without Statistics on a Nonindexed Column*

Statistics on Nonindexed Column	Figure	Cost	
		Avg. Duration (ms)	Number of Reads
With statistics	Figure 13-11	98	48
Without statistics	Figure 13-13	262	20273

The number of logical reads and the CPU utilization are higher with no statistics on the nonindexed columns. Without these statistics, the optimizer can't create a cost-effective plan because it effectively has to guess at the selectivity through a set of built-in heuristic calculations.

A query execution plan highlights the missing statistics by placing an exclamation point on the operator that would have used the statistics. You can see this in the clustered index scan operators in the previous execution plans (Figures 13-12 and 13-14), as well as in the detailed description in the Warnings section in the properties of a node in a graphical execution plan, as shown in Figure 13-15 for table `Test1`.



⊟ Warnings	Columns With No Statistics: [AdventureWorks2(
⊟ Columns With No Statistics	
⊟ Column Reference	[AdventureWorks2017].[dbo].[Test1].Test1_C2
Column	Test1_C2
Database	[AdventureWorks2017]
Schema	[dbo]
Table	[Test1]

Figure 13-15. *Missing statistics indication in a graphical plan*

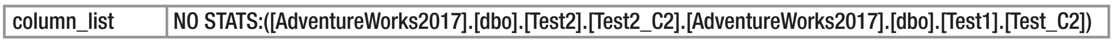
Note In a database application, there is always the possibility of queries using columns with no indexes. Therefore, in most systems, for performance reasons, leaving the auto create statistics feature of SQL Server databases on is strongly recommended.

You can query the plans in cache to identify those plans that may have missing statistics.

```
SELECT dest.text AS query,
       deqs.execution_count,
       deqp.query_plan
FROM sys.dm_exec_query_stats AS deqs
    CROSS APPLY sys.dm_exec_text_query_plan(deqs.plan_handle,
                                            deqs.statement_start_offset,
                                            deqs.statement_end_offset) AS
detqp
    CROSS APPLY sys.dm_exec_query_plan(deqs.plan_handle) AS deqp
    CROSS APPLY sys.dm_exec_sql_text(deqs.sql_handle) AS dest
WHERE detqp.query_plan LIKE '%ColumnsWithNoStatistics%';
```

This query cheats just a little bit. I'm using a wildcard on both sides of a variable with the LIKE operator, which is actually a common code issue (addressed in more detail in Chapter 20), but the alternative in this case is to run an XQuery, which requires loading the XML parser. Depending on the amount of memory available to your system, this approach, the wildcard search, can work a lot faster than querying the XML of the execution plan directly. Query tuning isn't just about using a single method but understanding how they all fit together.

If you are in a situation where you need to disable the automatic creation of statistics, you may still want to track where statistics may have been useful to your queries. You can use the Extended Events missing_column_statistics event to capture that information. For the previous examples, you can see an example of the output of this event in Figure 13-16.



column_list	NO STATS:([AdventureWorks2017].[dbo].[Test2].[Test2_C2].[AdventureWorks2017].[dbo].[Test1].[Test_C2])

Figure 13-16. *Output from missing_column_statistics Extended Events event*

The column_list will show which columns did not have statistics. You can then decide whether you want to create your own statistics to benefit the query in question.

Before proceeding, be sure to turn the automatic creation of statistics back on.

```
ALTER DATABASE AdventureWorks2017 SET AUTO_CREATE_STATISTICS ON;
```

Analyzing Statistics

Statistics are collections of information defined within three sets of data: the header, the density graph, and the histograms. One of the most commonly used of these data sets is the histogram. A *histogram* is a statistical construct that shows how often data falls into varying categories called *steps*. The histogram stored by SQL Server consists of a sampling of data distribution for a column or an index key (or the first column of a multicolumn index key) of up to 200 rows. The information on the range of index key values between two consecutive samples is one step. These steps consist of varying size intervals between the 200 values stored. A step provides the following information:

- The top value of a given step (RANGE_HI_KEY)
- The number of rows equal to RANGE_HI_KEY (EQ_ROWS)

- The number of rows between the previous top value and the current top value, without counting either of these boundary points (RANGE_ROWS)
- The number of distinct values in the range (DISTINCT_RANGE_ROWS); if all values in the range are unique, then RANGE_ROWS equals DISTINCT_RANGE_ROWS
- The average number of rows equal to any potential key value within a range (AVG_RANGE_ROWS)

For example, when referencing an index, the value of AVG_RANGE_ROWS for a key value within a step in the histogram helps the optimizer decide how (and whether) to use the index when the indexed column is referred to in a WHERE clause. Because the optimizer can perform a SEEK or SCAN operation to retrieve rows from a table, the optimizer can decide which operation to perform based on the number of potential matching rows for the index key value. This can be even more precise when referencing the RANGE_HI_KEY since the optimizer can know that it should find a fairly precise number of rows from that value (assuming the statistics are up-to-date).

To understand how the optimizer's data retrieval strategy depends on the number of matching rows, create a test table with different data distributions on an indexed column.

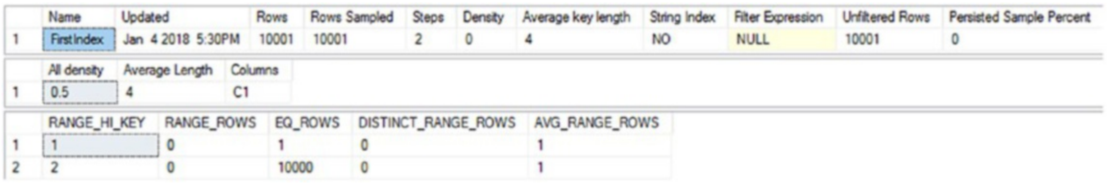
```
IF (SELECT  OBJECT_ID('dbo.Test1')
   ) IS NOT NULL
    DROP TABLE dbo.Test1 ;
GO

CREATE TABLE dbo.Test1 (C1 INT, C2 INT IDENTITY) ;

INSERT  INTO dbo.Test1
        (C1)
VALUES  (1) ;

SELECT TOP 10000
        IDENTITY( INT,1,1 ) AS n
INTO    #Nums
FROM    Master.dbo.SysColumns sc1,
        Master.dbo.SysColumns sc2 ;
```

```
INSERT  INTO dbo.Test1
        (C1)
        SELECT  2
        FROM    #Nums ;

DROP TABLE #Nums;

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

When the preceding nonclustered index is created, SQL Server automatically creates statistics on the index key. You can obtain statistics for this nonclustered index (`FirstIndex`) by executing the `DBCC SHOW_STATISTICS` command.

```
DBCC SHOW_STATISTICS(Test1, FirstIndex);
```

Figure 13-17 shows the statistics output.



	Name	Updated	Rows	Rows Sampled	Steps	Density	Average key length	String Index	Filter Expression	Unfiltered Rows	Persisted Sample Percent
1	FirstIndex	Jan 4 2018 5:30PM	10001	10001	2	0	4	NO	NULL	10001	0

	All density	Average Length	Columns
1	0.5	4	C1

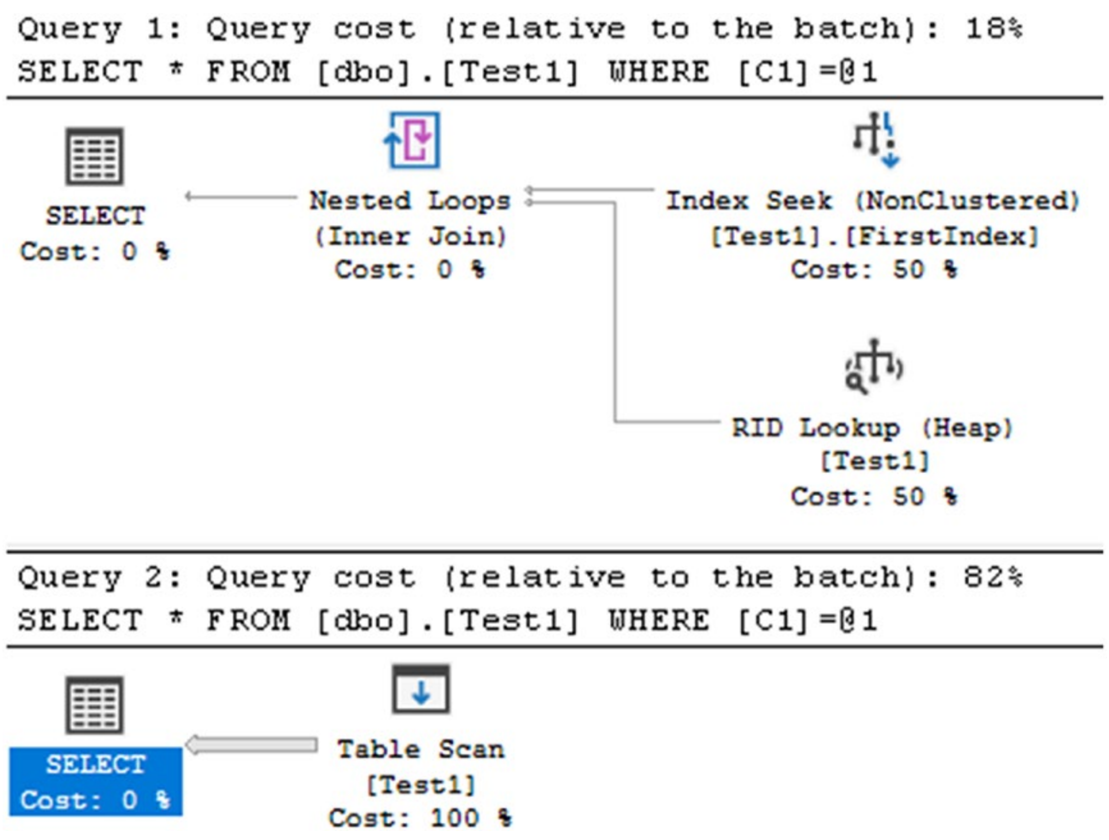
	RANGE_HI_KEY	RANGE_ROWS	EQ_ROWS	DISTINCT_RANGE_ROWS	AVG_RANGE_ROWS
1	1	0	1	0	1
2	2	0	10000	0	1

Figure 13-17. *Statistics on index FirstIndex*

Now, to understand how effectively the optimizer decides upon different data retrieval strategies based on statistics, execute the following two queries requesting a different number of rows:

```
--Retrieve 1 row;
SELECT  *
FROM    dbo.Test1
WHERE   C1 = 1;

--Retrieve 10000 rows;
SELECT  *
FROM    dbo.Test1
WHERE   C1 = 2;
```

Figure 13-18 shows execution plans of these queries.



```
Query 1: Query cost (relative to the batch): 18%
SELECT * FROM [dbo].[Test1] WHERE [C1]=@1
```

```
Query 2: Query cost (relative to the batch): 82%
SELECT * FROM [dbo].[Test1] WHERE [C1]=@1
```

Figure 13-18. *Execution plans of small and large result set queries*

From the statistics, the optimizer can find the number of rows needed for the preceding two queries. Understanding that there is only one row to be retrieved for the first query, the optimizer chose an `Index Seek` operation, followed by the necessary `RID Lookup` to retrieve the data not stored with the clustered index. For the second query, the optimizer knows that a large number of rows (10,000 rows) will be affected and therefore avoided the index to attempt to improve performance. (Chapter 8 explains indexing strategies in detail.)

Besides the information contained in the histogram, the header has other useful information including the following:

- The time statistics were last updated
- The number of rows in the table

- The average index key length
- The number of rows sampled for the histogram
- Densities for combinations of columns

Information on the time of the last update can help you decide whether you should manually update the statistics. The average key length represents the average size of the data in the index key columns. It helps you understand the width of the index key, which is an important measure in determining the effectiveness of the index. As explained in Chapter 6, a wide index might be costly to maintain and requires more disk space and memory pages but, as explained in the next section, can make an index extremely selective.

Density

When creating an execution plan, the query optimizer analyzes the statistics of the columns used in the filter and `JOIN` clauses. A filter criterion with high selectivity limits the number of rows from a table to a small result set and helps the optimizer keep the query cost low. A column with a unique index will have a high selectivity since it can limit the number of matching rows to one.

On the other hand, a filter criterion with low selectivity will return a large result set from the table. A filter criterion with low selectivity can make a nonclustered index on the column ineffective. Navigating through a nonclustered index to the base table for a large result set is usually costlier than scanning the base table (or clustered index) directly because of the cost overhead of lookups associated with the nonclustered index. You can observe this behavior in the first execution plan in Figure 13-18.

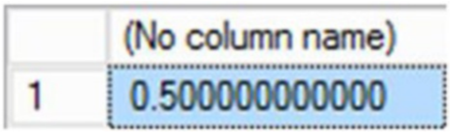
Statistics track the selectivity of a column in the form of a density ratio. A column with high selectivity (or uniqueness) will have low density. A column with low density (that is, high selectivity) is suitable for a filtering criteria because it can help the optimizer retrieve a small number of rows very fast. This is also the principle on which filtered indexes operate since the filter's goal is to increase the selectivity, or density, of the index.

Density can be expressed as follows:

```
Density = 1 / Number of distinct values for a column
```

Density will always come out as a number somewhere between 0 and 1. The lower the column density, the more suitable it is for use as an index key. You can perform your own calculations to determine the density of columns within your own indexes and statistics. For example, to calculate the density of column C1 from the test table built by the previous script, use the following (results in Figure 13-19):

```
SELECT 1.0 / COUNT(DISTINCT C1)
FROM dbo.Test1;
```



```
   (No column name)
1  0.500000000000
```

***Figure 13-19.** Results of density calculation for column C1*

You can see this as actual data in the `All density` column in the output from `DBCC SHOW_ STATISTICS`. This high-density value for the column makes it a less suitable candidate for an index, even a filtered index. However, the statistics of the index key values maintained in the steps help the query optimizer use the index for the predicate `C1 = 1`, as shown in the previous execution plan.

Statistics on a Multicolumn Index

In the case of an index with one column, statistics consist of a histogram and a density value for that column. Statistics for a composite index with multiple columns consist of one histogram for the first column only and multiple density values. This is one reason why it's generally a good practice to put the more selective column, the one with the lowest density, first when building a compound index or compound statistics. The density values include the density for the first column and for each additional combination of the index key columns. Multiple density values help the optimizer find the selectivity of the composite index when multiple columns are referred to by predicates in the `WHERE`, `HAVING`, and `JOIN` clauses. Although the first column can help determine the histogram, the final density of the column itself would be the same regardless of column order.

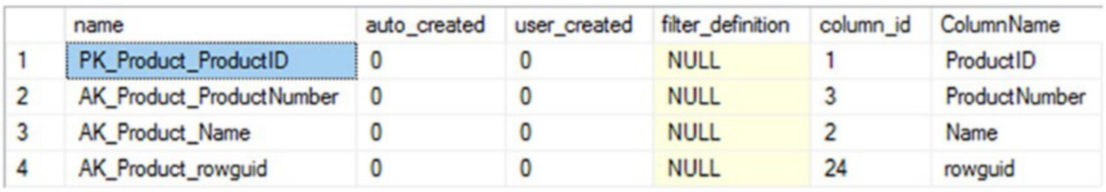
Multicolumn density graphs can come through multiple columns in the key of an index or from manually created statistics. But, you'll never see a multicolumn statistic, and subsequently a density graph, created by the automatic statistics creation process. Let's look at a quick example. Here's a query that could easily benefit from a set of statistics with two columns:

```
SELECT  p.Name,
        p.Class
FROM    Production.Product AS p
WHERE   p.Color = 'Red' AND
        p.DaysToManufacture > 15;
```

An index on the columns `p.Color` and `p.DaysToManufacture` would have a multicolumn density value. Before running this, here's a query that will let you just look at the basic construction of statistics on a given table:

```
SELECT s.name,
       s.auto_created,
       s.user_created,
       s.filter_definition,
       sc.column_id,
       c.name AS ColumnName
FROM sys.stats AS s
    JOIN sys.stats_columns AS sc
        ON sc.stats_id = s.stats_id
           AND sc.object_id = s.object_id
    JOIN sys.columns AS c
        ON c.column_id = sc.column_id
           AND c.object_id = s.object_id
WHERE s.object_id = OBJECT_ID('Production.Product');
```

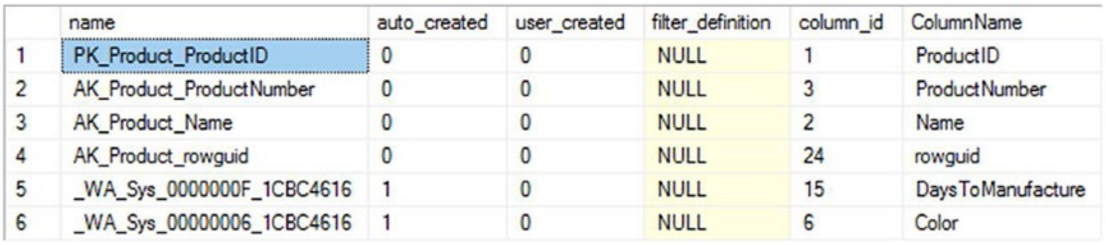
Running this query against the Production.Product table results in Figure 13-20.



	name	auto_created	user_created	filter_definition	column_id	ColumnName
1	PK_Product_ProductID	0	0	NULL	1	ProductID
2	AK_Product_ProductNumber	0	0	NULL	3	ProductNumber
3	AK_Product_Name	0	0	NULL	2	Name
4	AK_Product_rowguid	0	0	NULL	24	rowguid

Figure 13-20. *List of statistics for the Product table*

You can see the indexes on the table, and each one consists of a single column. Now I'll run the query that could benefit from a multicolumn density graph. But, rather than trying to track down the statistics information through SHOWSTATISTICS, I'll just query the system tables again. The results are in Figure 13-21.



	name	auto_created	user_created	filter_definition	column_id	ColumnName
1	PK_Product_ProductID	0	0	NULL	1	ProductID
2	AK_Product_ProductNumber	0	0	NULL	3	ProductNumber
3	AK_Product_Name	0	0	NULL	2	Name
4	AK_Product_rowguid	0	0	NULL	24	rowguid
5	_WA_Sys_0000000F_1CBC4616	1	0	NULL	15	DaysToManufacture
6	_WA_Sys_00000006_1CBC4616	1	0	NULL	6	Color

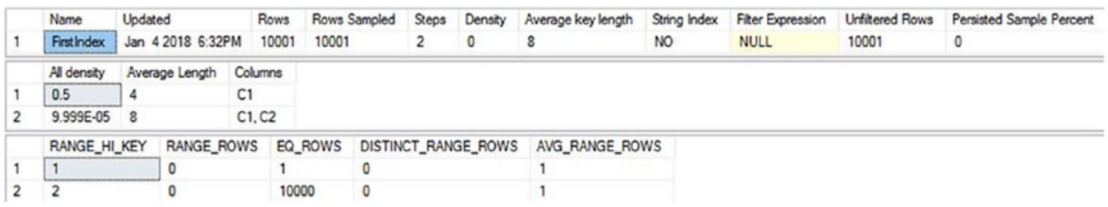
Figure 13-21. *Two new statistics have been added to the Product table*

As you can see, instead of adding a single statistic with multiple columns, two new statistics were created. You will get a multicolumn statistic only in a multicolumn index key or with manually created statistics.

To better understand the density values maintained for a multicolumn index, you can modify the nonclustered index used earlier to include two columns.

```
CREATE NONCLUSTERED INDEX FirstIndex
ON dbo.Test1
(
    C1,
    C2
)
WITH (DROP_EXISTING = ON);
```

Figure 13-22 shows the resultant statistics provided by DBCC SHOWSTATISTICS.



	Name	Updated	Rows	Rows Sampled	Steps	Density	Average key length	String Index	Filter Expression	Unfiltered Rows	Persisted Sample Percent
1	FirstIndex	Jan 4 2018 6:32PM	10001	10001	2	0	8	NO	NULL	10001	0

	All density	Average Length	Columns
1	0.5	4	C1
2	9.999E-05	8	C1, C2

	RANGE_HI_KEY	RANGE_ROWS	EQ_ROWS	DISTINCT_RANGE_ROWS	AVG_RANGE_ROWS
1	1	0	1	0	1
2	2	0	10000	0	1

***Figure 13-22.** Statistics on the multicolumn index FirstIndex*

As you can see, there are two density values under the All density column.

- The density of the first column
- The density of the (first + second) columns

For a multicolumn index with three columns, the statistics for the index would also contain the density value of the (first + second + third) columns. The histogram won't contain selectivity values for any other combination of columns. Therefore, this index (FirstIndex) won't be very useful for filtering rows only on the second column (C2) because that value of the second column (C2) alone isn't maintained in the histogram and, by itself, isn't part of the density graph.

You can compute the second density value (0.000099990000) shown in Figure 13-19 through the following steps. This is the number of distinct values for a column combination of (C1, C2).

```
SELECT 1.0 / COUNT(*)
FROM
(SELECT DISTINCT C1, C2 FROM dbo.Test1) AS DistinctRows;
```

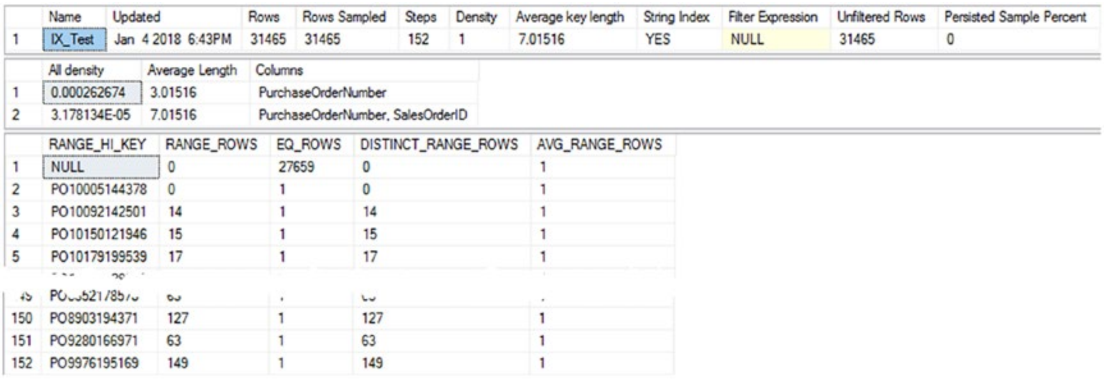
Statistics on a Filtered Index

The purpose of a filtered index is to limit the data that makes up the index and therefore change the density and histogram to make the index perform better. Instead of a test table, this example will use a table from the AdventureWorks2017 database. Create an index on the Sales.PurchaseOrderHeader table on the PurchaseOrderNumber column.

```
CREATE INDEX IX_Test ON Sales.SalesOrderHeader (PurchaseOrderNumber);
```

Figure 13-23 shows the header and the density of the output from `DBCC SHOWSTATISTICS` run against this new index.

```
DBCC SHOW_STATISTICS('Sales.SalesOrderHeader',IX_Test);
```



	Name	Updated	Rows	Rows Sampled	Steps	Density	Average key length	String Index	Filter Expression	Unfiltered Rows	Persisted Sample Percent
1	IX_Test	Jan 4 2018 6:43PM	31465	31465	152	1	7.01516	YES	NULL	31465	0

	All density	Average Length	Columns
1	0.000262674	3.01516	PurchaseOrderNumber
2	3.178134E-05	7.01516	PurchaseOrderNumber, SalesOrderID

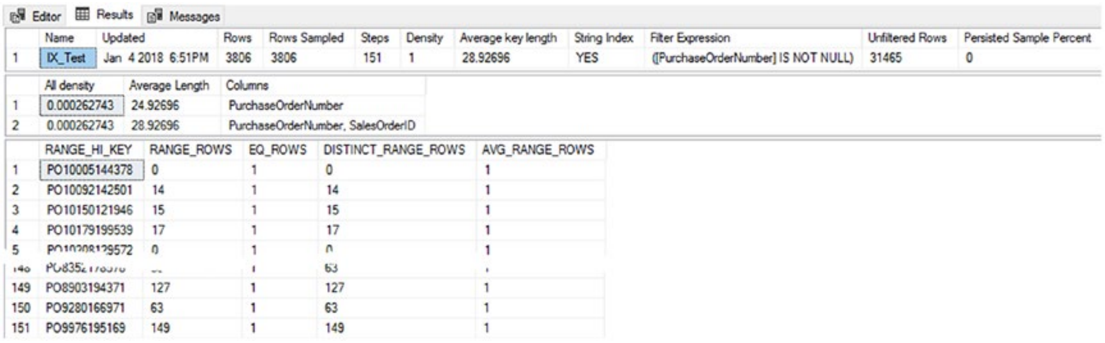
	RANGE_HI_KEY	RANGE_ROWS	EQ_ROWS	DISTINCT_RANGE_ROWS	AVG_RANGE_ROWS
1	NULL	0	27659	0	1
2	PO10005144378	0	1	0	1
3	PO10092142501	14	1	14	1
4	PO10150121946	15	1	15	1
5	PO10179199539	17	1	17	1
[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]
[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]
150	PO8903194371	127	1	127	1
151	PO9280166971	63	1	63	1
152	PO9976195169	149	1	149	1

***Figure 13-23.** Statistics header of an unfiltered index*

If the same index is re-created to deal with values of the column that are not `null`, it would look something like this:

```
CREATE INDEX IX_Test
ON Sales.SalesOrderHeader
(
    PurchaseOrderNumber
)
WHERE PurchaseOrderNumber IS NOT NULL
WITH (DROP_EXISTING = ON);
```

And now, in Figure 13-24, take a look at the statistics information.



Editor | Results | Messages

	Name	Updated	Rows	Rows Sampled	Steps	Density	Average key length	String Index	Filter Expression	Unfiltered Rows	Persisted Sample Percent
1	IX_Test	Jan 4 2018 6:51PM	3806	3806	151	1	28.92696	YES	([PurchaseOrderNumber] IS NOT NULL)	31465	0

	All density	Average Length	Columns
1	0.000262743	24.92696	PurchaseOrderNumber
2	0.000262743	28.92696	PurchaseOrderNumber, SalesOrderID

	RANGE_HI_KEY	RANGE_ROWS	EQ_ROWS	DISTINCT_RANGE_ROWS	AVG_RANGE_ROWS
1	PO10005144378	0	1	0	1
2	PO10092142501	14	1	14	1
3	PO10150121946	15	1	15	1
4	PO10179199539	17	1	17	1
5	PO10208129572	0	1	0	1
148	PO8352[illegible]	[illegible]	1	63	[illegible]
149	PO8903194371	127	1	127	1
150	PO9280166971	63	1	63	1
151	PO9976195169	149	1	149	1

Figure 13-24. *Statistics header for a filtered index*

First you can see that the number of rows that compose the statistics has radically dropped in the filtered index because there is a filter in place, from 31465 to 3806. Notice also that the average key length has increased since you're no longer dealing with zero-length strings. A filter expression has been defined rather than the `NULL` value visible in Figure 13-23. But the unfiltered rows of both sets of data are the same.

The density measurements are interesting. Notice that the density is close to the same for both values, but the filtered density is slightly lower, meaning fewer unique values. This is because the filtered data, while marginally less selective, is actually more accurate, eliminating all the empty values that won't contribute to a search. And the density of the second value, which represents the clustered index pointer, is identical with the value of the density of the `PurchaseOrderNumber` alone because each represents the same amount of unique data. The density of the additional clustered index in the previous column is a much smaller number because of all the unique values of `SalesOrderld` that are not included in the filtered data because of the elimination of the `NULL` values. You can also see the first column of the histogram shows a `NULL` value in Figure 13-23 but has a value in Figure 13-24.

One other option open to you is to create filtered statistics. This allows you to create even more fine-tuned histograms. This can be especially useful on partitioned tables. This is necessary because statistics are not automatically created on partitioned tables and you can't create your own using `CREATE STATISTICS`. You can create filtered indexes by partition and get statistics or create filtered statistics specifically by partition.

Before going on, clean the indexes created, if any.

```
DROP INDEX Sales.SalesOrderHeader.IX_Test;
```

Cardinality

The statistics, consisting of the histogram and density, are used by the query optimizer to calculate how many rows are to be expected by each operation within the execution of the query. This calculation to determine the number of rows returned is called the *cardinality estimate*. Cardinality represents the number of rows in a set of data, which means it's directly related to the density measures in SQL Server. Starting in SQL Server 2014, a different cardinality estimator is at work. This is the first change to the core cardinality estimation process since SQL Server 7.0. The changes to some areas of the estimator means that the optimizer reads from the statistics in the same way as previously, but the optimizer makes different kinds of calculations to determine the number of rows that are going to go through each operation in the execution plan depending on the cardinality calculations that have been modified.

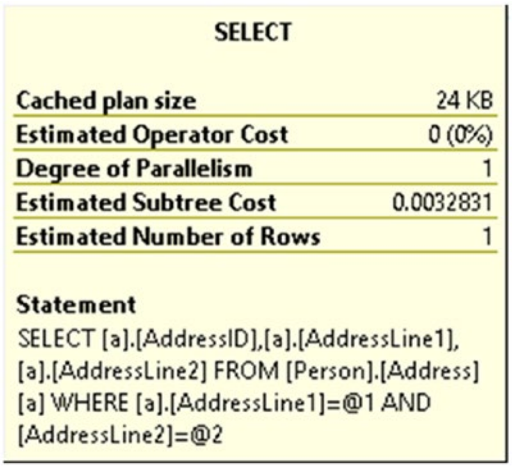
Before we discuss the details, let's see this in action. First, we'll change the cardinality estimation for the database to use the old estimator.

```
ALTER DATABASE SCOPED CONFIGURATION SET LEGACY_CARDINALITY_ESTIMATION = ON;
```

With that in place, I want to run a simple query.

```
SELECT a.AddressID,
       a.AddressLine1,
       a.AddressLine2
FROM Person.Address AS a
WHERE a.AddressLine1 = '5980 Icicle Circle'
      AND AddressLine2 = 'Unit H';
```

There's no need to explore the entire execution plan here. Instead, I want to look at the Estimated Row Count value on the SELECT operator, as shown in Figure 13-25.



SELECT

Cached plan size	24 KB
Estimated Operator Cost	0 (0%)
Degree of Parallelism	1
Estimated Subtree Cost	0.0032831
Estimated Number of Rows	1

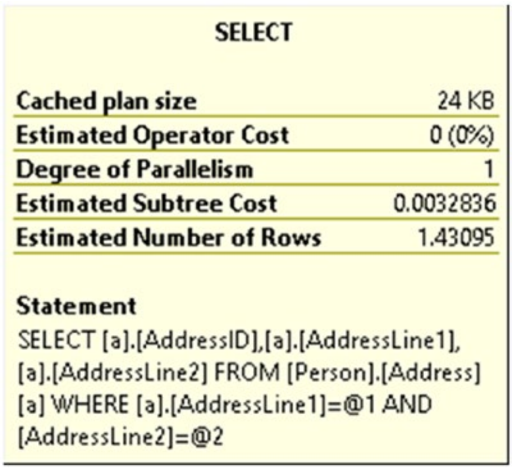
Statement
SELECT [a].[AddressID],[a].[AddressLine1], [a].[AddressLine2] FROM [Person].[Address] [a] WHERE [a].[AddressLine1]=@1 AND [AddressLine2]=@2

Figure 13-25. *Row counts with the old cardinality estimation engine*

You can see that the Estimated Number of Rows is equal to 1. Now, let's turn the legacy cardinality estimation back off.

```
ALTER DATABASE SCOPED CONFIGURATION SET LEGACY_CARDINALITY_ESTIMATION = OFF;
```

If we rerun the queries and take a look at the SELECT operator again, things have changed (see Figure 13-26).



SELECT

Cached plan size	24 KB
Estimated Operator Cost	0 (0%)
Degree of Parallelism	1
Estimated Subtree Cost	0.0032836
Estimated Number of Rows	1.43095

Statement
SELECT [a].[AddressID],[a].[AddressLine1], [a].[AddressLine2] FROM [Person].[Address] [a] WHERE [a].[AddressLine1]=@1 AND [AddressLine2]=@2

Figure 13-26. *Row counts with the modern cardinality estimation engine*

You can see that the estimated number of rows has changed from 1 to 1.43095. This is a direct reflection of the newer cardinality estimator.

Most of the time the data used to drive execution plans is pulled from the histogram. In the case of a single predicate, the values simply use the selectivity defined by the histogram. But, when multiple columns are used for filtering, the cardinality calculation has to take into account the potential selectivity of each column. Prior to SQL Server 2014, there were a couple of simple calculations used to determine cardinality. For an AND combination, the calculation was based on multiplying the selectivity of the first column by the selectivity of the second, something like this:

```
Selectivity1 * Selectivity2 * Selectivity3 ...
```

An OR calculation between two columns was more complex. The new AND calculation looks like this:

```
Selectivity1 * Power(Selectivity2,1/2) * Power(Selectivity3,1/4) ...
```

In short, instead of simply multiplying the selectivity of each column to make the overall selectivity more and more selective, a different calculation is supplied, going from the least selective to the most selective data but arriving at a softer, less skewed estimate by getting the power of one-half the selectivity, then one-quarter, and then one-eighth, and so on, depending on how many columns of data are involved. The working assumption is that data isn't one set of columns with no relation to the next set; instead, there is a correlation between the data, making a certain degree of duplication possible. This new calculation won't change all execution plans generated, but the potentially more accurate estimates could change them in some locations. When an OR clause is used, the calculations have again changed to suggest the possibility of correlation between columns.

In the previous example, we did see exactly that. There were three rows returned, and the 1.4 row estimate is closer than the 1 row estimate to that value of 3.

Starting in SQL Server 2014 with a compatibility level of 120, even more new calculations are taking place. This means that for most queries, on average, you may see performance enhancements if your statistics are up-to-date because having more accurate cardinality calculations means the optimizer will make better choices. But, you may also see performance degradation with some queries because of the changes in the way cardinality is calculated. This is to be expected because of the wide variety of workloads, schemas, and data distributions that you may encounter.

Another new cardinality estimation assumption changed in SQL Server 2014. In SQL Server 2012 and earlier, when a value in an index that consisted of an increasing or decreasing increment, such as an identity column or a datetime value, introduced a new row that fell outside the existing histogram, the optimizer would fall back on its default estimate for data without statistics, which was one row. This could lead to seriously inaccurate query plans, causing poor performance. Now, there are all new calculations.

First, if you have created statistics using a `FULLSCAN`, explained in detail in the “Statistics Maintenance” section, and there have been no modifications to the data, then the cardinality estimation works the same as it did before. But, if the statistics have been created with a default sampling or data has been modified, then the cardinality estimator works off the average number of rows returned within that set of statistics and assumes that value instead of a single row. This can make for much more accurate execution plans, but assuming only a reasonably consistent distribution of data. An uneven distribution, referred to as *skewed data*, can lead to bad cardinality estimations that can result in behavior similar to bad parameter sniffing, covered in detail in Chapter 18.

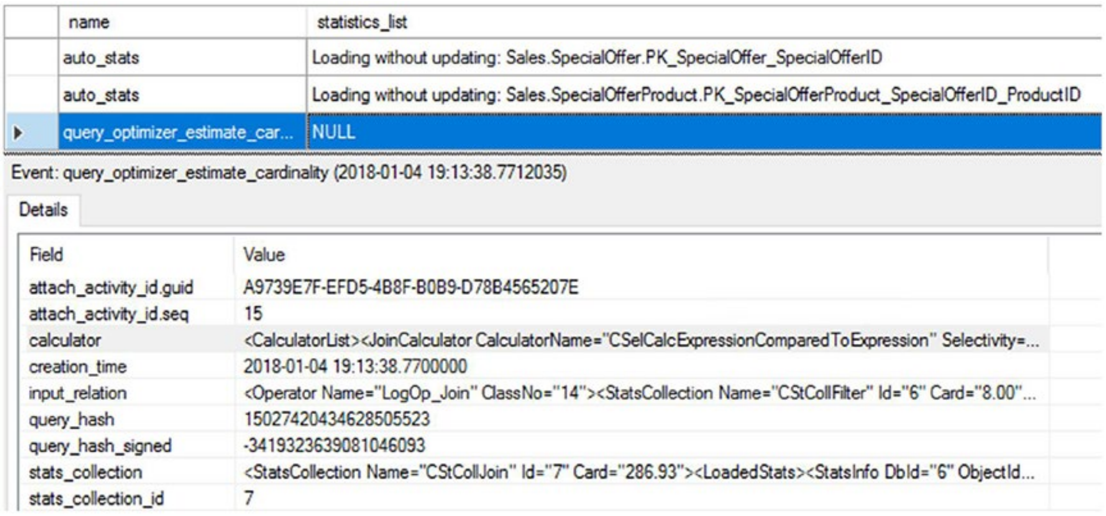
You can now observe cardinality estimations in action using Extended Events with the event `query_optimizer_estimate_cardinality`. I won't go into all the details of every possible output from the events, but I do want to show how you can observe optimizer behavior and correlate it between execution plans and the cardinality estimations. For the vast majority of query tuning, this won't be all that helpful, but if you're unsure of how the optimizer is making the estimates that it does or if those estimates seem inaccurate, you can use this method to further investigate the information.

Note The `query_optimizer_estimate_cardinality` event is in the Debug package within Extended Events. The debug events are primarily for internal use at Microsoft. The events contained within Debug, including `query_optimizer_estimate_cardinality`, are subject to change or removal without notice.

First, you should set up an Extended Events session with the `query_optimizer_estimate_cardinality` event. I've created an example including the `auto_stats` and `sql_batch_complete` events. Then, I ran a query.

```
SELECT  so.Description,
        p.Name AS ProductName,
        p.ListPrice,
        p.Size,
        pv.AverageLeadTime,
        pv.MaxOrderQty,
        v.Name AS VendorName
FROM    Sales.SpecialOffer AS so
JOIN    Sales.SpecialOfferProduct AS sop
ON      sop.SpecialOfferID = so.SpecialOfferID
JOIN    Production.Product AS p
ON      p.ProductID = sop.ProductID
JOIN    Purchasing.ProductVendor AS pv
ON      pv.ProductID = p.ProductID
JOIN    Purchasing.Vendor AS v
ON      v.BusinessEntityID = pv.BusinessEntityID
WHERE so.DiscountPct > .15;
```

I chose a query that's a little complex so that there are plenty of operators in the execution plan. When I run the query, I can then see the output of the Extended Events session, as shown in Figure 13-27.



name	statistics_list
auto_stats	Loading without updating: Sales.SpecialOffer.PK_SpecialOffer_SpecialOfferID
auto_stats	Loading without updating: Sales.SpecialOfferProduct.PK_SpecialOfferProduct_SpecialOfferID_ProductID
query_optimizer_estimate_car...	NULL

Event: query_optimizer_estimate_cardinality (2018-01-04 19:13:38.7712035)

Details

Field	Value
attach_activity_id.guid	A9739E7F-EFD5-4B8F-B0B9-D78B4565207E
attach_activity_id.seq	15
calculator	<CalculatorList><JoinCalculator CalculatorName="CSelCalcExpressionComparedToExpression" Selectivity=...
creation_time	2018-01-04 19:13:38.7700000
input_relation	<Operator Name="LogOp_Join" ClassNo="14"><StatsCollection Name="CStCollFilter" Id="6" Card="8.00"...
query_hash	15027420434628505523
query_hash_signed	-3419323639081046093
stats_collection	<StatsCollection Name="CStCollJoin" Id="7" Card="286.93"><LoadedStats><StatsInfo DbId="6" ObjectId...
stats_collection_id	7

Figure 13-27. *Session showing output from the query_optimizer_estimate_cardinality event*

The first two events visible in Figure 13-27 show the auto_stats event firing where it loaded the statistics for two columns; Sales.SpecialOffer.PK_SpecailOffer_SpecialOfferID and Sales.SpecialOfferProduct.PK_SpecialOfferProduct_SpecialOfferID_ProductID. This means the statistics were readied prior to the cardinality estimation calculation firing. The information on the Details tab is the output from the cardinality estimation calculation. The detailed information is contained as JSON in the calculator, input_relation, and stats_collection fields. These will show the types of calculations and the values used in those calculations. For example, here is the output from the calculator field in Figure 13-27:

```
<CalculatorList>
  <JoinCalculator CalculatorName="CSelCalcExpre
ssionComparedToExpression" Selectivity="0.067"
SelectivityBeforeAdjustmentForOverPopulatedDimension="0.063" />
</CalculatorList>
```

While the calculations themselves are not always clear, you can see the values that are being used by the calculation and where they are coming from. In this case, the calculation is comparing two values and arriving at a new selectivity based on that calculation.

At the bottom of Figure 13-27 you can see the stats_collection_id value, which, in this case, is 7. You can use this value to track down some of the calculations within an execution plan to understand both what the calculation is doing and how it is used.

We're going to first capture the execution plan. Even if you are retrieving the plan from the Query Store or some other source, the stats_collection_id values are stored with the plan. Once you have a plan, we can take advantage of new functionality within SSMS 2017. Right-clicking within a graphical plan will open a context menu, as shown in Figure 13-28.

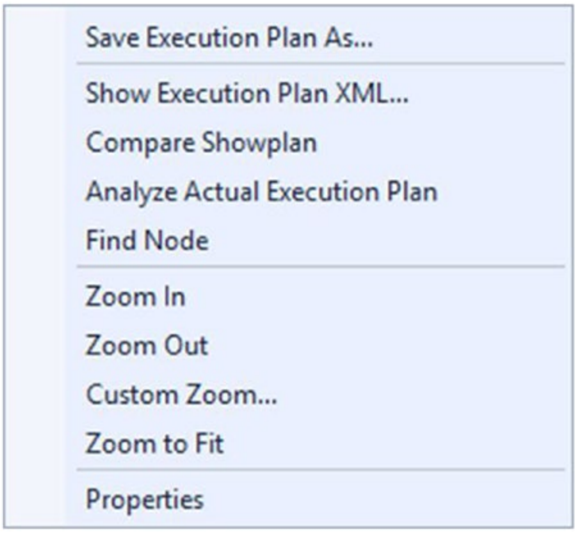
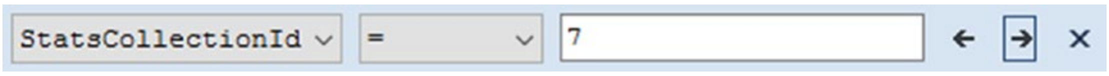

Save Execution Plan As...
Show Execution Plan XML...
Compare Showplan
Analyze Actual Execution Plan
Find Node
Zoom In
Zoom Out
Custom Zoom...
Zoom to Fit
Properties

Figure 13-28. *Execution plan context menu showing the Find Node menu selection*

What we want to do is use the Find Node command to search through the execution plan. Clicking that menu choice will open a small window at the top of the execution plan, which I've filled out in Figure 13-29.



StatsCollectionId = 7

Figure 13-29. *The Find Node interface within a graphical execution plan*

I've selected the execution plan property that I'm interested in, `StatsCollectionId`, and provided the value from the extended event shown in Figure 13-27. When I then click the arrows, this will take me directly to the node that has a matching value for this property and select it. With this, I can combine the information gathered by the extended event with the information within the execution plan to arrive a better understanding of how the optimizer is consuming the statistics.

Finally, in SQL Server Management Studio 2017, you can also get a listing of the statistics that were specifically used by the optimizer to put together the execution plan. In the first operator, in this case a `SELECT` operator, within the properties, you can get a complete listing of all statistics similar to what you can see in Figure 13-30.