When run, this query results in a Key Lookup operation (Figure 12-15) and the following I/O:

```
Duration: 251 mc
Reads: 10
```

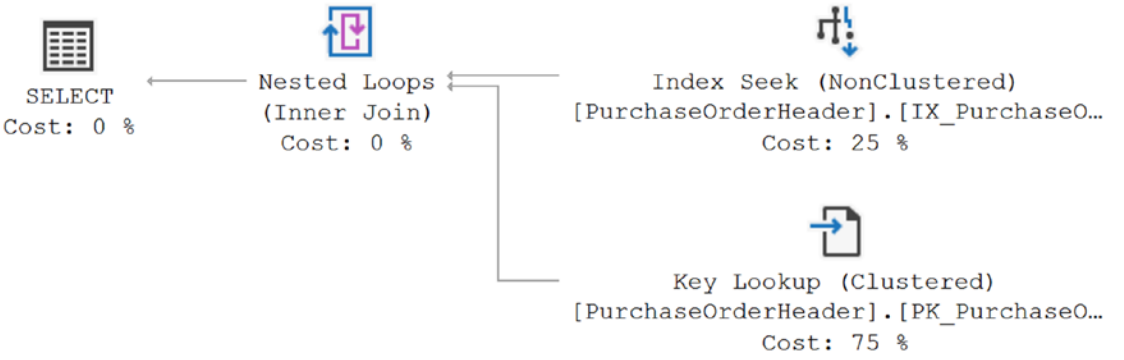


Figure 12-15. *A Key Lookup operation*

The lookup is caused since all the columns referred to by the SELECT statement and WHERE clause are not included in the nonclustered index on column VendorID. Using the nonclustered index is still better than not using it since that would require a scan on the table (in this case, a clustered index scan) with a larger number of logical reads.

To avoid the lookup, you can consider a covering index on the column OrderDate, as explained in the previous section. But in addition to the covering index solution, you can consider an index join. As you learned, an index join requires narrower indexes than the covering index and thereby provides the following two benefits:

- Multiple narrow indexes can serve a larger number of queries than the wide covering index.
- Narrow indexes require less maintenance overhead than the wide covering index.

To avoid the lookup using an index join, create a narrow nonclustered index on column OrderDate that is not included in the existing nonclustered index.

```
CREATE NONCLUSTERED INDEX IX_TEST
ON Purchasing.PurchaseOrderHeader
(
    OrderDate
);
```

If you run the SELECT statement again, the following output and the execution plan shown in Figure 12-16 are returned:

```
Duration: 219 mc
Reads: 4
```

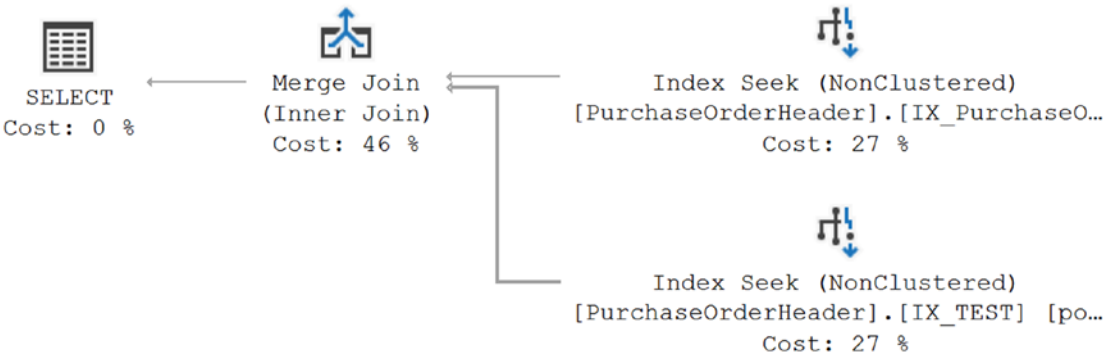


Figure 12-16. *Execution plan without a lookup*

From the preceding execution plan, you can see that the optimizer used the nonclustered index, IX_PurchaseOrder_VendorID, on column VendorID and the new nonclustered index, IX_TEST, on column OrderID to serve the query fully without hitting the storage location of the rest of the data. This index join operation avoided the lookup and consequently decreased the number of logical reads from 10 to 4.

It is true that a covering index on columns VendorID and OrderID could reduce the number of logical reads further. But it may not always be possible to use covering indexes since they can be wide and have their associated overhead. In such cases, an index join can be a good alternative.

Summary

As demonstrated in this chapter, the lookup step associated with a nonclustered index can make data retrieval through a nonclustered index very costly. The SQL Server optimizer takes this into account when generating an execution plan, and if it finds the overhead cost of using a nonclustered index to be high, it discards the index and performs a table scan (or a clustered index scan if the table is stored as a clustered index). Therefore, to improve the effectiveness of a nonclustered index, it makes sense

to analyze the cause of a lookup and consider whether you can avoid it completely by adding fields to the index key or to the `INCLUDE` column (or index join) and creating a covering index.

Up to this point, you have concentrated on indexing techniques and presumed that the SQL Server optimizer would be able to determine the effectiveness of an index for a query. In the next chapter, you will see the importance of statistics in helping the optimizer determine the effectiveness of an index.

CHAPTER 13

Statistics, Data Distribution, and Cardinality

By now, you should have a good understanding of the importance of indexes. But, the index alone is not what the optimizer uses to determine how it's going to access data. It also takes advantage of enforced referential constraint and other table structures. Finally, and possibly most important, the optimizer must have information about the data that defines an index or a column. That information is referred to as a *statistic*. Statistics define both the distribution of data and the uniqueness or selectivity of the data. Statistics are maintained both on indexes and on columns within the system. You can even define statistics manually yourself.

In this chapter, you'll learn the importance of statistics in query optimization. Specifically, I will cover the following topics:

- The role of statistics in query optimization
- The importance of statistics on columns with indexes
- The importance of statistics on nonindexed columns used in join and filter criteria
- Analysis of single-column and multicolumn statistics, including the computation of selectivity of a column for indexing
- Statistics maintenance
- Effective evaluation of statistics used in query execution

G. Fritchey, *SQL Server 2017 Query Performance Tuning*, https://doi.org/10.1007/978-1-4842-3888-2_13

The Role of Statistics in Query Optimization

SQL Server's query optimizer is a cost-based optimizer; it decides on the best data access mechanism and join strategy by identifying the selectivity, how unique the data is, and which columns are used in filtering the data (meaning via the `WHERE`, `HAVING`, or `JOIN` clause). Statistics are automatically created with an index, but they also exist on columns without an index that are used as part of a predicate. As you learned in Chapter 7, a nonclustered index is a great way to retrieve data that is covered by the index, whereas with queries that need columns outside the key, a clustered index can work better. With a large result set, going to the clustered index or table directly is usually more beneficial.

Up-to-date information on data distribution in the columns referenced as predicates helps the optimizer determine the query strategy to use. In SQL Server, this information is maintained in the form of statistics, which are essential for the cost-based optimizer to create an effective query execution plan. Through the statistics, the optimizer can make reasonably accurate estimates about how long it will take to return a result set or an intermediate result set and therefore determine the most effective operations to use to efficiently retrieve or modify the data as defined by the T-SQL statement. As long as you ensure that the default statistical settings for the database are set, the optimizer will be able to do its best to determine effective processing strategies dynamically. Also, as a safety measure while troubleshooting performance, you should ensure that the automatic statistics maintenance routine is doing its job as desired. Where necessary, you may even have to take manual control over the creation and/or maintenance of statistics. (I cover this in the "Manual Maintenance" section, and I cover the precise nature of the functions and shape of statistics in the "Analyzing Statistics" section.) In the following section, I show you why statistics are important to indexed columns and nonindexed columns functioning as predicates.

Statistics on an Indexed Column

The usefulness of an index is largely dependent on the statistics of the indexed columns; without statistics, SQL Server's cost-based query optimizer can't decide upon the most effective way of using an index. To meet this requirement, SQL Server automatically creates the statistics of an index key whenever the index is created. It isn't possible to turn this feature off. This occurs for both rowstore and columnstore indexes.

As data changes, the data retrieval mechanism required to keep the cost of a query low may also change. For example, if a table has only one matching row for a certain column

value, then it makes sense to retrieve the matching rows from the table by going through the nonclustered index on the column. But if the data in the table changes so that a large number of rows are added with the same column value, then using the nonclustered index may no longer make sense. To be able to have SQL Server decide this change in processing strategy as the data changes over time, it is vital to have up-to-date statistics.

SQL Server can keep the statistics on an index updated as the contents of the indexed column are modified. By default, this feature is turned on and is configurable through the Properties ➤ Options ➤ Auto Update Statistics setting of a database. Updating statistics consumes extra CPU cycles and associated I/O. To optimize the update process, SQL Server uses an efficient algorithm detailed in the "Automatic Maintenance" section.

This built-in intelligence keeps the CPU utilization by each process low. It's also possible to update the statistics asynchronously. This means when a query would normally cause statistics to be updated, instead that query proceeds with the old statistics, and the statistics are updated offline. This can speed up the response time of some queries, such as when the database is large or when you have a short timeout period. It may also slow performance if the changes in statistics are enough to warrant a radical change in the plan.

You can manually disable (or enable) the auto update statistics and the auto update statistics asynchronously features by using the `ALTER DATABASE` command. By default, the auto update statistics feature and the auto creation feature are enabled, and it is strongly recommended that you keep them enabled. The auto update statistics asynchronously feature is disabled by default. Turn this feature on only if you've determined it will help with timeouts or waits caused by statistics updates.

Note I explain `ALTER DATABASE` later in this chapter in the "Manual Maintenance" section.

Benefits of Updated Statistics

The benefits of performing an auto update usually outweigh its cost on the system resources for the majority of systems. If you have large tables (and I mean hundreds of gigabytes for a single table), you may be in a situation where letting the statistics update automatically is less beneficial. In this case, you may want to try using the sliding scale supplied through trace flag 2371, or you may be in a situation where automatic statistics

maintenance doesn't work well. However, this is an extreme edge case, and even here, you may find that an auto update of the statistics doesn't negatively impact your system.

To more directly control the behavior of the data, instead of using the tables in AdventureWorks2017 for this set of examples, you will create one manually. Specifically, create a test table with only three rows and a nonclustered index.

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

SELECT TOP 1500
    IDENTITY(INT, 1, 1) AS n
INTO #Nums
FROM master.dbo.syscolumns AS sC1,
     master.dbo.syscolumns AS sC2;

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

DROP TABLE #Nums;


CREATE NONCLUSTERED INDEX i1 ON dbo.Test1 (C1);
```

If you execute a SELECT statement with a selective filter criterion on the indexed column to retrieve only one row, as shown in the following line of code, then the optimizer uses a nonclustered index seek, as shown in the execution plan in Figure 13-1:

```
SELECT *
FROM dbo.Test1
WHERE C1 = 2;
```

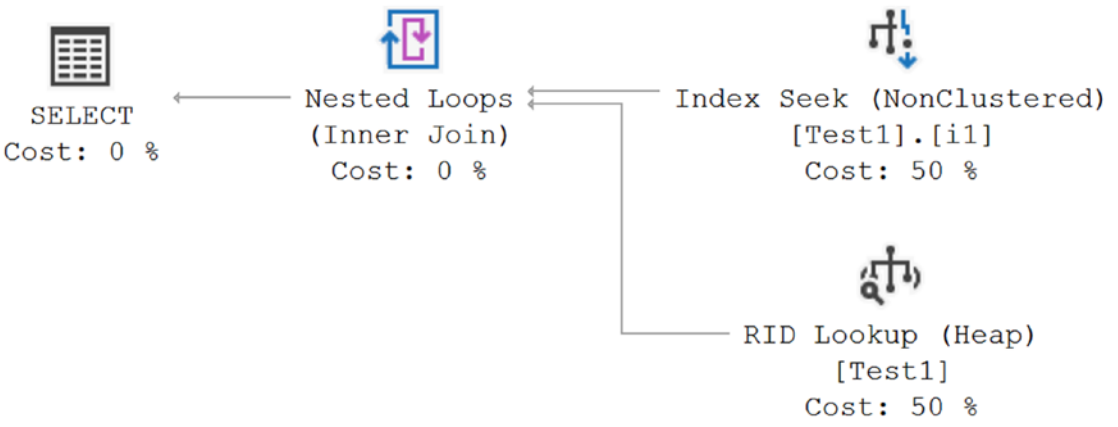


Figure 13-1. *Execution plan for a small result set*

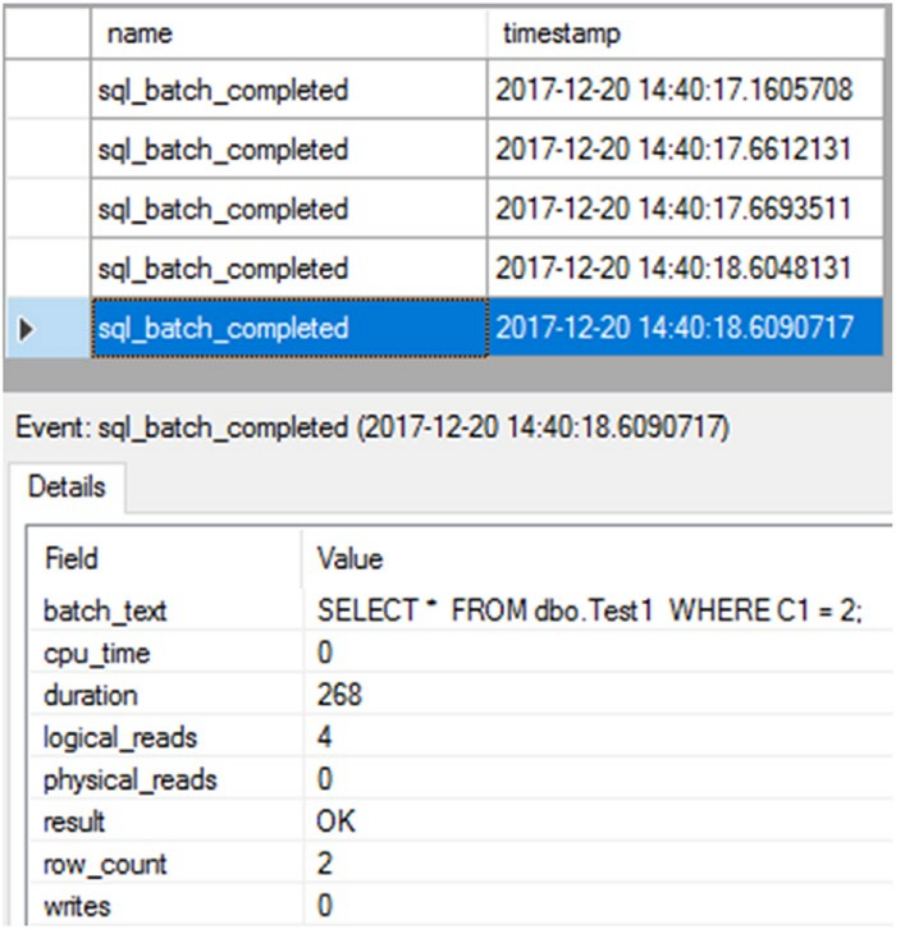
To understand the effect of small data modifications on a statistics update, create a session using Extended Events. In the session, add the event `auto_stats`, which captures statistics update and create events, and add `sql_batch_completed`. Here's the script to create and start an Extended Events session:

```
CREATE EVENT SESSION [Statistics]
ON SERVER
    ADD EVENT sqlserver.auto_stats
    (ACTION (sqlserver.sql_text)
     WHERE (sqlserver.database_name = N'AdventureWorks2017')),
    ADD EVENT sqlserver.sql_batch_completed
    (WHERE (sqlserver.database_name = N'AdventureWorks2017'));
GO
ALTER EVENT SESSION [Statistics] ON SERVER STATE = START;
GO
```

Add only one row to the table.

```
INSERT  INTO dbo.Test1
        (C1)
VALUES  (2);
```

When you reexecute the preceding SELECT statement, you get the same execution plan as shown in Figure 13-1. Figure 13-2 shows the events generated by the SELECT query.



	name	timestamp
	sql_batch_completed	2017-12-20 14:40:17.1605708
	sql_batch_completed	2017-12-20 14:40:17.6612131
	sql_batch_completed	2017-12-20 14:40:17.6693511
	sql_batch_completed	2017-12-20 14:40:18.6048131
▶	sql_batch_completed	2017-12-20 14:40:18.6090717

Event: sql_batch_completed (2017-12-20 14:40:18.6090717)

Details

Field	Value
batch_text	SELECT * FROM dbo.Test1 WHERE C1 = 2;
cpu_time	0
duration	268
logical_reads	4
physical_reads	0
result	OK
row_count	2
writes	0

Figure 13-2. *Session output after the addition of a small number of rows*

The session output doesn't contain any activity representing a statistics update because the number of changes fell below the threshold where any table that has more than 500 rows must have 20 percent of the number of rows be added, modified, or removed, or, using the newer behavior, doesn't reflect adequate scaled changes.

To understand the effect of large data modification on statistics update, add 1,500 rows to the table.

```
SELECT TOP 1500
    IDENTITY(INT, 1, 1) AS n
INTO #Nums
FROM master.dbo.syscolumns AS scl,
     master.dbo.syscolumns AS sC2;
INSERT INTO dbo.Test1 (C1)
SELECT 2
FROM #Nums;
DROP TABLE #Nums;
```

Now, if you reexecute the SELECT statement, like so, a large result set (1,502 rows out of 3,001 rows) will be retrieved:

```
SELECT  *
FROM    dbo.Test1
WHERE   C1 = 2;
```

Since a large result set is requested, scanning the base table directly is preferable to going through the nonclustered index to the base table 1,502 times. Accessing the base table directly will prevent the overhead cost of bookmark lookups associated with the nonclustered index. This is represented in the resultant execution plan (see Figure 13-3).

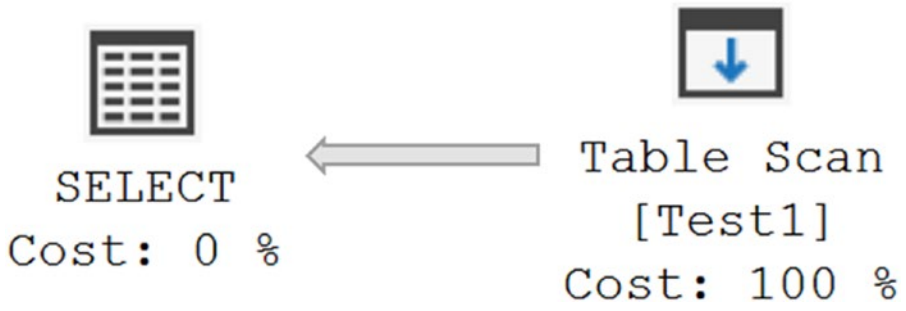
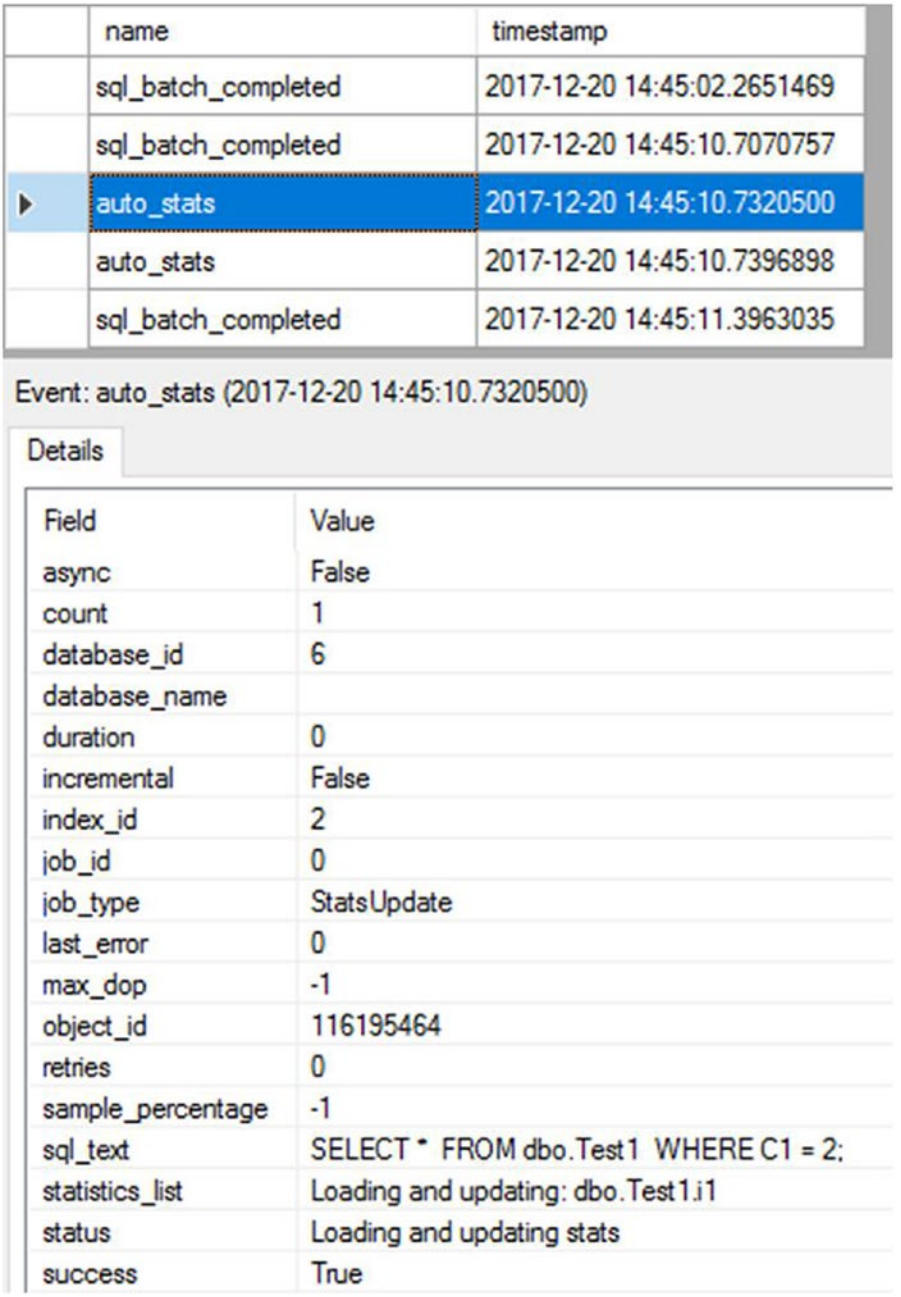


Figure 13-3. *Execution plan for a large result set*

Figure 13-4 shows the resultant session output.



	name	timestamp
	sql_batch_completed	2017-12-20 14:45:02.2651469
	sql_batch_completed	2017-12-20 14:45:10.7070757
▶	auto_stats	2017-12-20 14:45:10.7320500
	auto_stats	2017-12-20 14:45:10.7396898
	sql_batch_completed	2017-12-20 14:45:11.3963035

Event: auto_stats (2017-12-20 14:45:10.7320500)

Details

Field	Value
async	False
count	1
database_id	6
database_name	
duration	0
incremental	False
index_id	2
job_id	0
job_type	StatsUpdate
last_error	0
max_dop	-1
object_id	116195464
retries	0
sample_percentage	-1
sql_text	SELECT * FROM dbo.Test1 WHERE C1 = 2;
statistics_list	Loading and updating: dbo.Test1.i1
status	Loading and updating stats
success	True

Figure 13-4. *Session output after the addition of a large number of rows*

The session output includes multiple `auto_stats` events since the threshold was exceeded by the large-scale update this time. You can tell what each of the events is doing by looking at the details. Figure 13-4 shows the `job_type` value, in this case `StatsUpdate`. You'll also see the statistics that are being updated listed in the `statistics_list` column. Another point of interest is the Status column, which can tell you more about what part of the statistics update process is occurring, in this case "Loading and update stats." The second `auto_stats` event visible in Figure 13-4 shows a `statistics_list` value of "Updated: dbo.Test1.i1" indicating that the update process was complete. You can then see immediately following that `auto_stats` event the `sql_batch_completed` event of the query itself. These activities consume some extra CPU cycles to get the stats up-to-date. However, by doing this, the optimizer determines a better data-processing strategy and keeps the overall cost of the query low. The resulting change to a more efficient execution plan, the `Table Scan` operation of Figure 13-3, is why automatic update of statistics is so desirable. This also illustrates how an asynchronous update of statistics could potentially cause problems because the query would have executed with the old, less efficient execution plan.

Drawbacks of Outdated Statistics

As explained in the preceding section, the auto update statistics feature allows the optimizer to decide on an efficient processing strategy for a query as the data changes. If the statistics become outdated, however, then the processing strategies decided on by the optimizer may not be applicable for the current data set and thereby will degrade performance.

To understand the detrimental effect of having outdated statistics, follow these steps:

1. Re-create the preceding test table with 1,500 rows only and the corresponding nonclustered index.

2. Prevent SQL Server from updating statistics automatically as the data changes. To do so, disable the auto update statistics feature by executing the following SQL statement:

   ```
   ALTER DATABASE AdventureWorks2017 SET AUTO_UPDATE_
   STATISTICS OFF;
   ```

3. Add 1,500 rows to the table like before.

Now, reexecute the SELECT statement to understand the effect of the outdated statistics on the query optimizer. The query is repeated here for clarity:

```
SELECT *
FROM dbo.Test1
WHERE C1 = 2;
```

Figure 13-5 and Figure 13-6 show the resultant execution plan and the session output for this query, respectively.

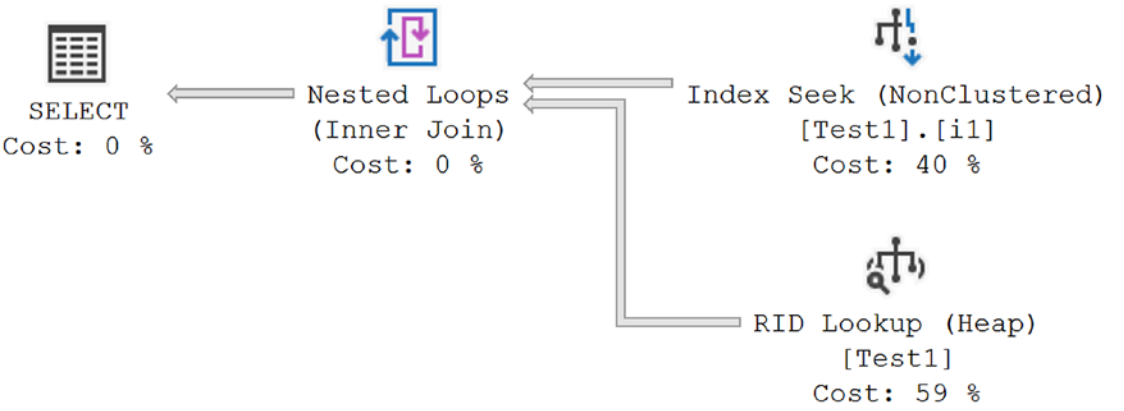


Figure 13-5. *Execution plan with AUTO_UPDATE_STATISTICS OFF*

Event: sql_batch_completed (2017-12-20 15:01:25.4888444)

Details

Field	Value
batch_text	SELECT * FROM dbo.Test1 WHERE C1 = 2;
cpu_time	0
duration	677640
logical_reads	1514
physical_reads	0
result	OK
row_count	1501
writes	2

Figure 13-6. *Session output details with AUTO_UPDATE_STATISTICS OFF*

With the auto update statistics feature switched off, the query optimizer has selected a different execution plan from the one it selected with this feature on. Based on the outdated statistics, which have only one row for the filter criterion (`C1 = 2`), the optimizer decided to use a nonclustered index seek. The optimizer couldn't make its decision based on the current data distribution in the column. For performance reasons, it would have been better to access the base table directly instead of going through the nonclustered index since a large result set (1,501 rows out of 3,000 rows) is requested.

You can see that turning off the auto update statistics feature has a negative effect on performance by comparing the cost of this query with and without updated statistics. Table 13-1 shows the difference in the cost of this query.

Table 13-1. *Cost of the Query with and Without Updated Statistics*

Statistics Update Status	Figure	Cost	
		Duration (ms)	Number of Reads
Updated	Figure 13-4	171	9
Not updated	Figure 13-6	678	1510

The number of reads and the duration are significantly higher when the statistics are out-of-date, even though the data returned is identical and the query was precisely the same. Therefore, it is recommended that you keep the auto update statistics feature on. The benefits of keeping statistics updated usually outweigh the costs of performing the update. Before you leave this section, turn `AUTO_UPDATE_STATISTICS` back on (although you can also manually update statistics if you choose).

```
ALTER DATABASE AdventureWorks2017 SET AUTO_UPDATE_STATISTICS ON;
```

Statistics on a Nonindexed Column

Sometimes you may have columns in join or filter criteria without any index. Even for such nonindexed columns, the query optimizer is more likely to make a better choice if it knows the cardinality and data distribution, the *statistics*, of those columns. Cardinality is the number of objects in a set, in this case rows. Data distribution would be how unique the overall set of data we're working with is.

In addition to statistics on indexes, SQL Server can build statistics on columns with no indexes. The information on data distribution, or the likelihood of a particular value occurring in a nonindexed column, can help the query optimizer determine an optimal processing strategy. This benefits the query optimizer even if it can't use an index to actually locate the values. SQL Server automatically builds statistics on nonindexed columns if it deems this information valuable in creating a better plan, usually when the columns are used in a predicate. By default, this feature is turned on, and it's configurable through the Properties ➤ Options ➤ Auto Create Statistics setting of a database. You can override this setting programmatically by using the `ALTER DATABASE` command. However, for better performance, it is strongly recommended that you keep this feature on.

One of the scenarios in which you may consider disabling this feature is while executing a series of ad hoc T-SQL activities that you will never execute again. Another is when you determine that a static, stable, but possibly not adequate set of statistics works better than the best possible set of statistics, but they may lead to uneven performance because of changing data distribution. Even in such a case, you should test whether you're better off paying the cost of automatic statistics creation to get a better plan in this one case as compared to affecting the performance of other SQL Server activities. For most systems, you should keep this feature on and not be concerned about it unless you see clear evidence of statistics creation causing performance issues.

Benefits of Statistics on a Nonindexed Column

To understand the benefit of having statistics on a column with no index, create two test tables with disproportionate data distributions, as shown in the following code. Both tables contain 10,001 rows. Table `Test1` contains only one row for a value of the second column (`Test1_C2`) equal to 1, and the remaining 10,000 rows contain this column value as 2. Table `Test2` contains exactly the opposite data distribution.

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

CREATE TABLE dbo.Test1 (Test1_C1 INT IDENTITY,
                        Test1_C2 INT);
```

```
INSERT INTO dbo.Test1 (Test1_C2)
VALUES (1);

SELECT TOP 10000
       IDENTITY(INT, 1, 1) AS n
INTO #Nums
FROM master.dbo.syscolumns AS scl,
     master.dbo.syscolumns AS sC2;

INSERT INTO dbo.Test1 (Test1_C2)
SELECT 2
FROM #Nums
GO

CREATE CLUSTERED INDEX i1 ON dbo.Test1 (Test1_C1)


--Create second table with 10001 rows, -- but opposite data distribution
IF (SELECT OBJECT_ID('dbo.Test2')) IS NOT NULL
    DROP TABLE dbo.Test2;
GO

CREATE TABLE dbo.Test2 (Test2_C1 INT IDENTITY,
                        Test2_C2 INT);

INSERT INTO dbo.Test2 (Test2_C2)
VALUES (2);

INSERT INTO dbo.Test2 (Test2_C2)
SELECT 1
FROM #Nums;
DROP TABLE #Nums;
GO

CREATE CLUSTERED INDEX il ON dbo.Test2 (Test2_C1);
```

Table 13-2 illustrates how the tables will look.

Table 13-2. *Sample Tables*

	Table Test1		Table Test2	
Column	Test1_c1	Test1_C2	Test2_c1	Test2_C2
Row1	1	1	1	2
Row2	2	2	2	1
RowN	N	2	N	1
Row10001	10001	2	10001	1

To understand the importance of statistics on a nonindexed column, use the default setting for the auto create statistics feature. By default, this feature is on. You can verify this using the DATABASEPROPERTYEX function (although you can also query the sys.databases view).

```
SELECT DATABASEPROPERTYEX('AdventureWorks2017',
                          'IsAutoCreateStatistics');
```

Note You can find a detailed description of configuring the auto create statistics feature later in this chapter.

Use the following SELECT statement to access a large result set from table Test1 and a small result set from table Test2. Table Test1 has 10,000 rows for the column value of Test1_C2 = 2, and table Test2 has 1 row for Test2_C2 = 2. Note that these columns used in the join and filter criteria have no index on either table.

```
SELECT t1.Test1_C2,
       t2.Test2_C2
FROM dbo.Test1 AS t1
    JOIN dbo.Test2 AS t2
        ON t1.Test1_C2 = t2.Test2_C2
WHERE t1.Test1_C2 = 2;
```

Figure 13-7 shows the actual execution plan for this query.

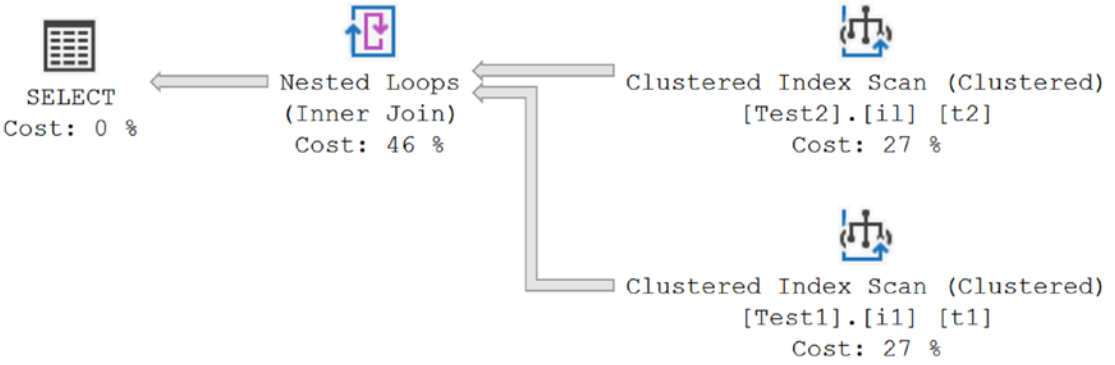


Figure 13-7. *Execution plan with AUTO_CREATE_STATISTICS ON*

Figure 13-8 shows the session output of the `auto_stats` event caused by this query. You can use this to evaluate some of the added costs for a given query.

name	attach_activity_id.seq	job_type	statistics_list	duration
auto_stats	1	StatsUpdate	Created: Test1_C2	11124
auto_stats	2	StatsUpdate	Loading without updating: dbo.Test1._WA_Sys_...	0
auto_stats	3	StatsUpdate	Created: Test2_C2	9065
auto_stats	4	StatsUpdate	Loading without updating: dbo.Test2._WA_Sys_...	0
sql_batch_completed	5	NULL	NULL	236480

Figure 13-8. *Extended Events session output with AUTO_CREATE_STATISTICS ON*

The session output shown in Figure 13-8 includes four `auto_stats` events creating statistics on the nonindexed columns referred to in the `JOIN` and `WHERE` clauses, `Test2_C2` and `Test1_C2`, and then loading those statistics for use inside the optimizer. This activity consumes a few extra CPU cycles (since no statistics could be detected) and took about 20,000 microseconds (mc), or 20ms. However, by consuming these extra CPU cycles, the optimizer decides upon a better processing strategy for keeping the overall cost of the query low.

To verify the statistics automatically created by SQL Server on the nonindexed columns of each table, run this SELECT statement against the sys.stats table.

```
SELECT s.name,
       s.auto_created,
       s.user_created
FROM sys.stats AS s
WHERE object_id = OBJECT_ID('Test1');
```

Figure 13-9 shows the automatic statistics created for table Test1.

	name	auto_created	user_created
1	i1	0	0
2	_WA_Sys_00000002_20ACD28B	1	0

Figure 13-9. *Automatic statistics for table Test1*

The statistics named _WA_SYS* are system-generated column statistics. You can tell this both by the name of the statistic and by the auto_created value, which, in this case, is equal to 1, whereas that same value for the index, i1, is 0. This is interesting since statistics created for indexes are also automatically created, but they're not considered part of the AUTO_CREATE_STATISTICS process since statistics on indexes will always be created.

To verify how a different result set size from the two tables influences the decision of the query optimizer, modify the filter criteria of the query to access an opposite result set size from the two tables (small from Test1 and large from Test2). Instead of filtering on Test1.Test1_C2 = 2, change it to filter on 1.

```
SELECT t1.Test1_C2,
       t2.Test2_C2
FROM dbo.Test1 AS t1
    JOIN dbo.Test2 AS t2
        ON t1.Test1_C2 = t2.Test2_C2
WHERE t1.Test1_C2 = 1;
```

Figure 13-10 shows the resultant execution plan, and Figure 13-11 shows the Extended Events session output of this query.

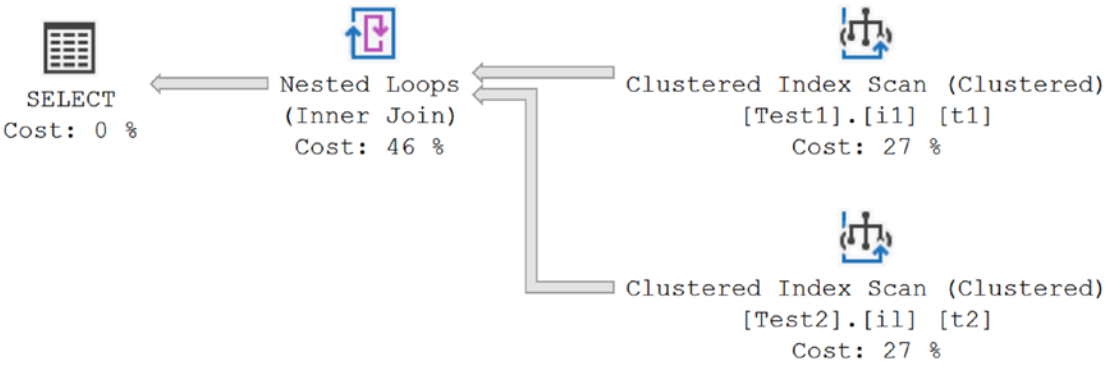


Figure 13-10. *Execution plan for a different result set*

Field	Value
attach_activity_id.guid	A61F2A34-75CA-422B-9789-9090B76E6868
attach_activity_id.seq	1
attach_activity_id_xfer.guid	3832346F-E305-4BB6-9534-C610B61071CA
attach_activity_id_xfer.seq	0
batch_text	SELECT t1.Test1_C2, t2.Test2_C2 FROM dbo.Test1 A...
cpu_time	0
duration	367754
logical_reads	48
physical_reads	0
result	OK
row_count	10000
writes	0

Figure 13-11. *Extended Events output for a different result set*

The resultant session output doesn't perform any additional SQL activities to manage statistics. The statistics on the nonindexed columns (`Test1.Test1_C2` and `Test2.Test2_C2`) had already been created when the indexes themselves were created and updated as the data changed.

For effective cost optimization, in each case the query optimizer selected different processing strategies, depending upon the statistics on the nonindexed

columns (`Test1.Test1_C2` and `Test2.Test2_C2`). You can see this from the previous two execution plans. In the first, table `Test1Test1` is the outer table for the nested loop join, whereas in the latest one, table `Test2` is the outer table. By having statistics on the nonindexed columns (`Test1.Test1_C2` and `Test2.Test2_C2`), the query optimizer can create a cost-effective plan suitable for each case.

An even better solution would be to have an index on the column. This would not only create the statistics on the column but also allow fast data retrieval through an `Index Seek` operation, while retrieving a small result set. However, in the case of a database application with queries referring to nonindexed columns in the `WHERE` clause, keeping the auto create statistics feature on still allows the optimizer to determine the best processing strategy for the existing data distribution in the column.

If you need to know which column or columns might be covered by a given statistic, you need to look into the `sys.stats_columns` system table. You can query it in the same way as you did the `sys.stats` table.

```
SELECT  *
FROM    sys.stats_columns
WHERE   object_id = OBJECT_ID('Test1');
```

This will show the column being referenced by the automatically created statistics. You can use this information to help you if you decide you need to create an index to replace the statistics because you will need to know which columns to create the index on. The column listed here is the ordinal position of the column within the table. To see the column name, you'd need to modify the query.

```
SELECT c.name,
       sc.object_id,
       sc.stats_column_id,
       sc.stats_id
FROM sys.stats_columns AS sc
    JOIN sys.columns AS c
        ON c.object_id = sc.object_id
           AND c.column_id = sc.column_id
WHERE sc.object_id = OBJECT_ID('Test1');
```

Drawback of Missing Statistics on a Nonindexed Column

To understand the detrimental effect of not having statistics on nonindexed columns, drop the statistics automatically created by SQL Server and prevent SQL Server from automatically creating statistics on columns with no index by following these steps:

1. Drop the automatic statistics created on column `Test1.Test1_C2` using the following SQL command, substituting the system name automatically given the statistics for the phrase `StatisticsName`:

   ```
   DROP STATISTICS [Test1].StatisticsName;
   ```

2. Similarly, drop the corresponding statistics on column `Test2.Test2_C2`.

3. Disable the auto create statistics feature by deselecting the Auto Create Statistics check box for the corresponding database or by executing the following SQL command:

   ```
   ALTER DATABASE AdventureWorks2017 SET AUTO_CREATE_
   STATISTICS OFF;
   ```

Now reexecute the `SELECT` statement `--nonindexed_select`.

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

Figure 13-12 and Figure 13-13 show the resultant execution plan and Extended Events output, respectively.

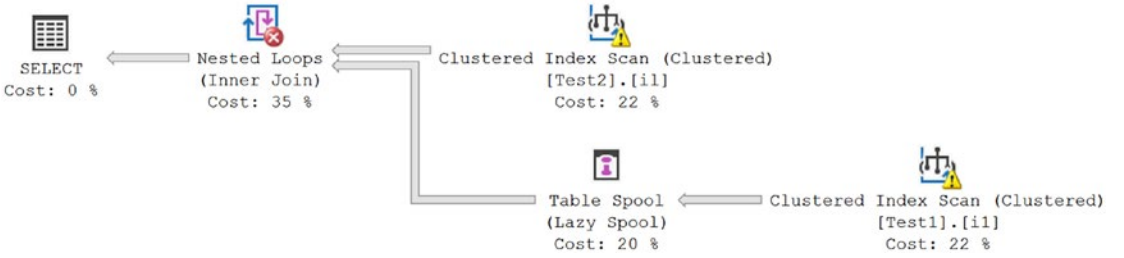


Figure 13-12. *Execution plan with AUTO_CREATE_STATISTICS OFF*

Field	Value
attach_activity_id.guid	5E051D9E-3D4A-459A-91DC-6D3D49E1F625
attach_activity_id.seq	1
attach_activity_id_xfer.guid	3832346F-E305-4BB6-9534-C610B61071CA
attach_activity_id_xfer.seq	0
batch_text	SELECT Test1.Test1_C2, Test2.Test2_C...
cpu_time	32000
duration	92960
logical_reads	20235
physical_reads	0
result	OK
row_count	10000
writes	27

Figure 13-13. *Trace output with AUTO_CREATE_STATISTICS OFF*

With the auto create statistics feature off, the query optimizer selected a different execution plan compared to the one it selected with the auto create statistics feature on. On not finding statistics on the relevant columns, the optimizer chose the first table (`Test1`) in the `FROM` clause as the outer table of the nested loop join operation. The optimizer couldn't make its decision based on the actual data distribution in the column. You can see the warning, an exclamation point, in the execution plan,

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;
```

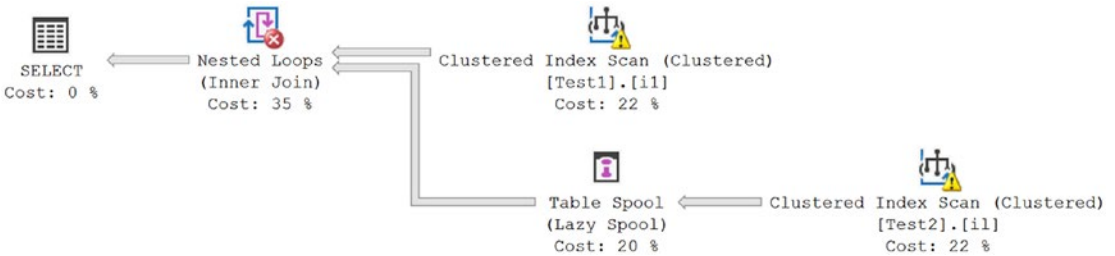


Figure 13-14. *Execution plan with AUTO_CREATE_STATISTICS OFF (a variation)*

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`.

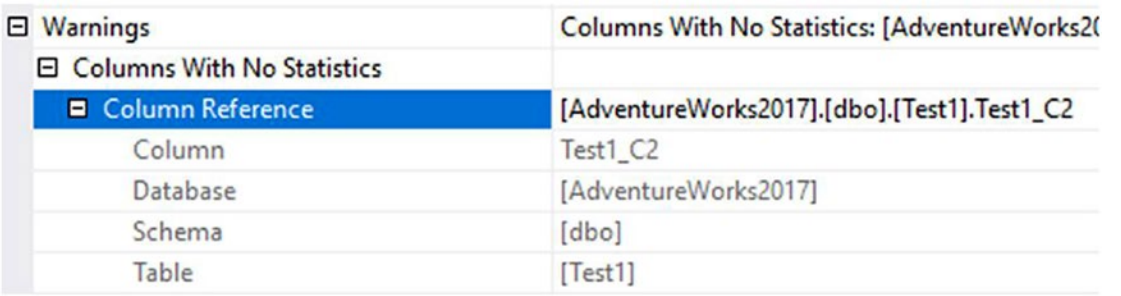


Figure 13-15. *Missing statistics indication in a graphical plan*

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.

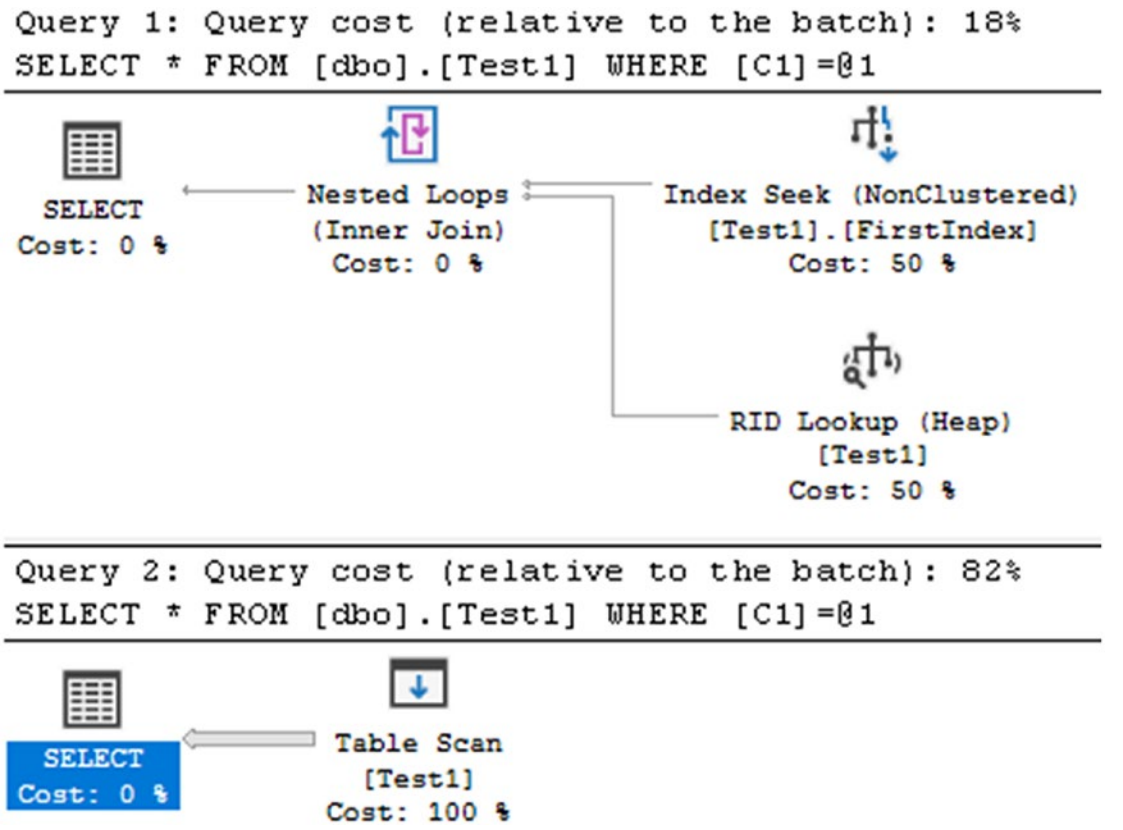


Figure 13-18. *Execution plans of small and large result set queries*

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

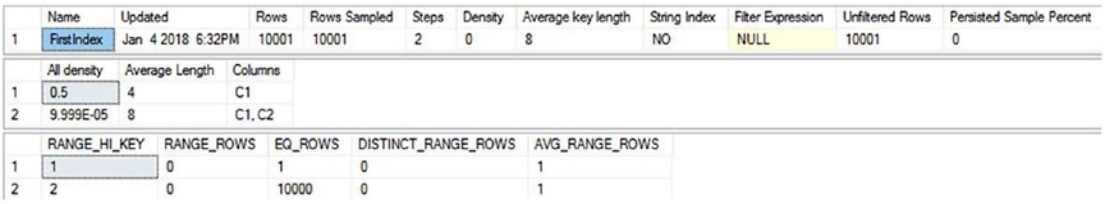
Figure 13-21. *Two new statistics have been added to the Product table*

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;
```

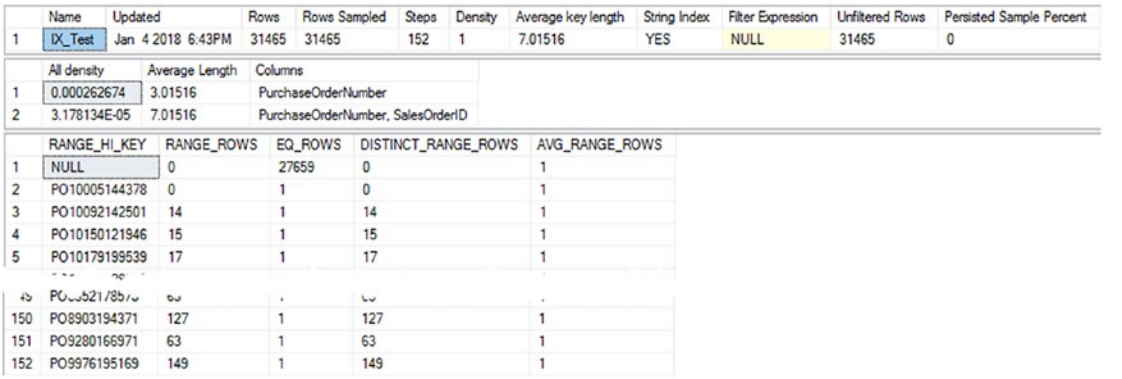
Statistics on a Filtered Index

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

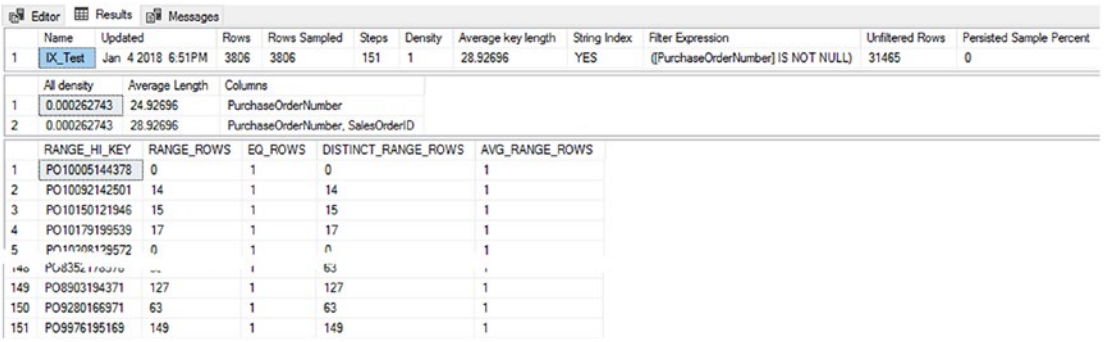
	RANGE_HI_KEY	RANGE_ROWS	EQ_ROWS	DISTINCT_RANGE_ROWS	AVG_RANGE_ROWS
1	NULL	0	27659	0	1
2	PO10005144378	0	1	0	1
3	PO10092142501	14	1	14	1
4	PO10150121946	15	1	15	1
5	PO10179199539	17	1	17	1
150	PO8903194371	127	1	127	1
151	PO9280166971	63	1	63	1
152	PO9976195169	149	1	149	1

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
146	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 `SalesOrderId` 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.

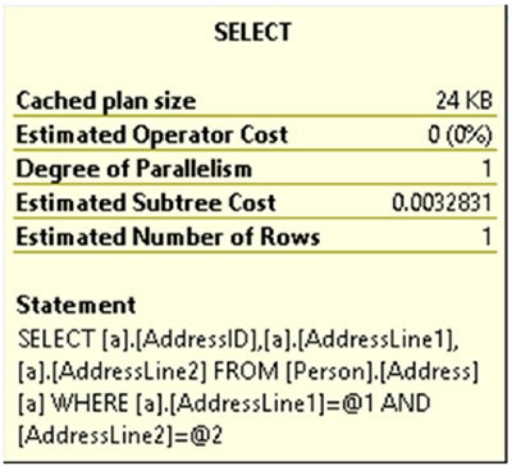


Figure 13-25. *Row counts with the old cardinality estimation engine*

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).

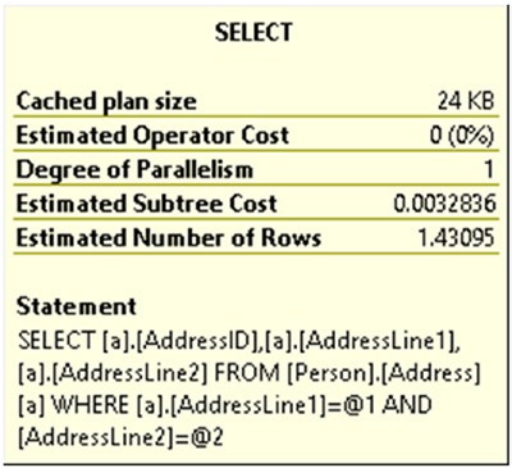


Figure 13-26. *Row counts with the modern cardinality estimation engine*

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.

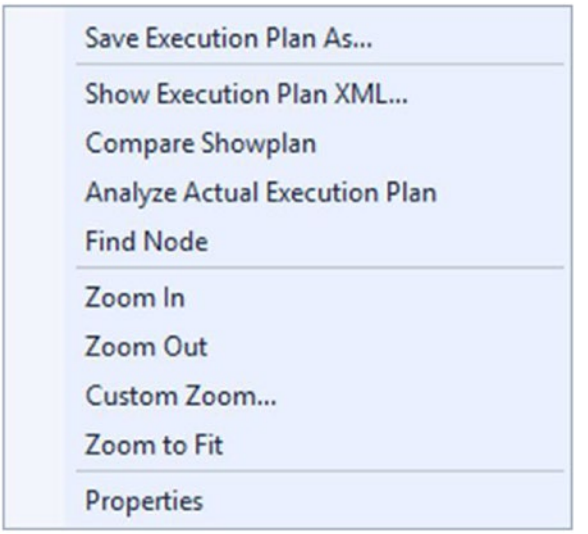


Figure 13-28. *Execution plan context menu showing the Find Node menu selection*

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.

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.

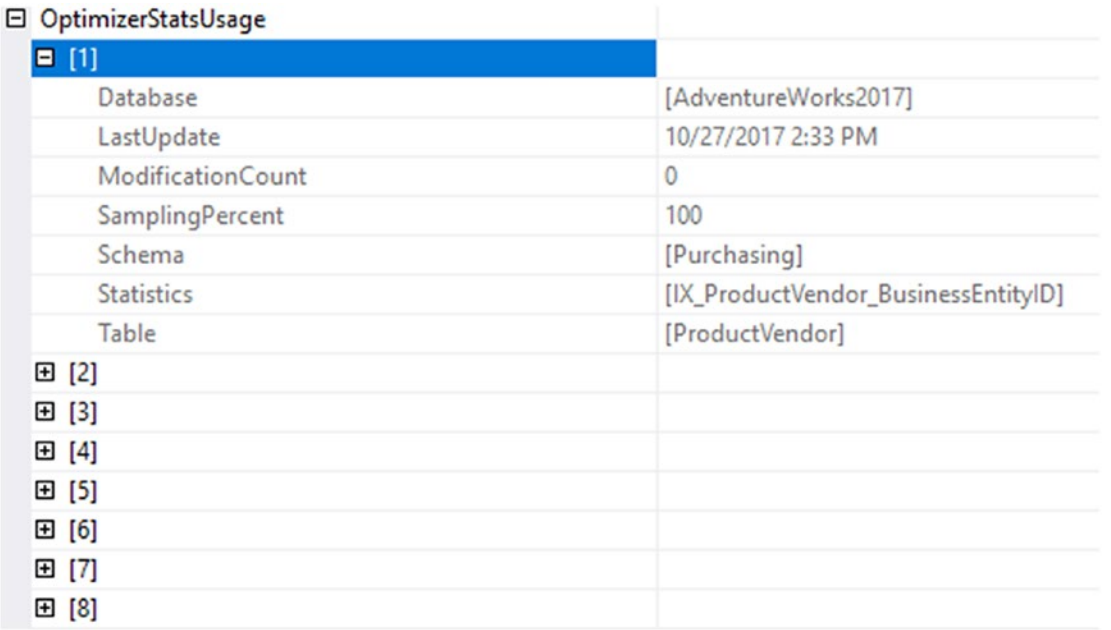


Figure 13-30. *Statistics in use within the execution plan generated for a query*

Enabling and Disabling the Cardinality Estimator

If you create a database in SQL Server 2014 or greater, it's going to automatically come with the compatibility level set to 120, or greater, which is the correct version for the latest SQL Server. But, if you restore or attach a database from a previous version of SQL Server, the compatibility level will be set to that version, 110 or before. That database will then use the SQL Server 7 cardinality estimator. You can tell this by looking at the execution plan in the first operator (`SELECT/INSERT/UPDATE/DELETE`) at the properties for the `CardinalityEstimationModelVersion`, as shown in Figure 13-31.

Figure 13-31. *Property in the first operator showing the cardinality estimator in use*

The value shown for SQL Server 2014–2017 will correspond to the version, 120, 130, 140. That's how you can tell what version of the cardinality estimator is in use. This is important because since the estimates can lead to changes in execution plans, it's really

important that you understand how to troubleshoot the issues in the event that you get a degradation in performance caused by the new cardinality estimations.

If you suspect that you are experiencing problems from the upgrade, you should absolutely compare your actual rows returned to the estimated rows returned in the operations within the execution plan. That's always a great way to determine whether statistics or cardinality estimations are causing you issues. You should be using the Query Store for both testing your upgrades and as part of the upgrade process (as outlined in Chapter 11). The Query Store is the best way to capture before and after the change in the cardinality estimation engine and the best way to deal with the individual queries that may go wrong.

You have the option of disabling the new cardinality estimation functionality by setting the compatibility level to 110, but that also disables other newer SQL Server functionality, so it might not be a good choice. You can run a trace flag against the restore of the database using `OPTION (QUERYTRACEON 9481)`; you'll target just the cardinality estimator for that database. If you determine in a given query that you're having issues with the new cardinality estimator, you can take advantage of trace flags in the query in the same way.

```
SELECT p.Name,
       p.Class
FROM Production.Product AS p
WHERE p.Color = 'Red'
      AND p.DaysToManufacture > 15
OPTION (QUERYTRACEON 9481);
```

Conversely, if you have turned off the cardinality estimator using the trace flag or compatibility level, you can selectively turn it on for a given query using the same functionality as earlier but substituting 2312 for the trace flag value.

Finally, a new function was introduced in SQL Server 2016, Database Scoped Configuration. Among other settings (which we'll discuss in appropriate places throughout the book), you can disable just the cardinality estimation engine without disabling all the modern functionality. The new syntax looks like this:

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

Using this command, you can change the behavior of the database without changing all other behaviors. You can also use the same command to turn off the legacy cardinality estimator. You also have the option of a USE hint on individual queries. Setting FORCE_LEGACY_CARDINALITY_ESTIMATION inside a query hint will make that query use the old cardinality estimation, and only that one query. This is probably the single safest option, although it does involve code changes.

Statistics DMOs

Prior to SQL Server 2016, the only way to get information on statistics was to use DBCC SHOW_STATISTICS. However, a couple of new DMFs have been introduced that can be useful. The sys.dm_db_stats_properties function returns the header information of a set of statistics. This means you quickly pull information out of the header. For example, use this query to retrieve when the statistics were last updated:

```
SELECT ddsp.object_id,
       ddsp.stats_id,
       ddsp.last_updated
FROM sys.dm_db_stats_properties(OBJECT_ID('HumanResources.Employee'),
                                2) AS ddsp;
```

The function requires that you pass the object_id that you're interested in and the statistics_id for that object. In this example we look at the column statistics on the HumanResources.Employee table.

The other function is sys.dm_db_stats_histogram. It works much the same way, allowing us to treat the histogram of statistics as a queryable object. For example, suppose we wanted to find a particular set of values within the histogram. Normally, you look for the range_hi_key value and then see whether the value you're looking for is less than one range_high_key but greater than another. It's entirely possible to automate this now.

```
WITH histo
AS (SELECT ddsh.step_number,
           ddsh.range_high_key,
           ddsh.range_rows,
           ddsh.equal_rows,
           ddsh.average_range_rows
```

```
    FROM sys.dm_db_stats_histogram(OBJECT_ID('HumanResources.Employee'),
                                   1) AS ddsh ),
     histojoin
AS (SELECT h1.step_number,
           h1.range_high_key,
           h2.range_high_key AS range_high_key_step1,
           h1.range_rows,
           h1.equal_rows,
           h1.average_range_rows
    FROM histo AS h1
        LEFT JOIN histo AS h2
            ON h1.step_number = h2.step_number + 1)
SELECT hj.range_high_key,
       hj.equal_rows,
       hj.average_range_rows
FROM histojoin AS hj
WHERE hj.range_high_key >= 17
      AND (   hj.range_high_key_step1 < 17
              OR hj.range_high_key_step1 IS NULL);
```

This query will look through the statistics in question on the `HumanResources.Employee` table and will find which row in the histogram would contain the value of 17.

Statistics Maintenance

SQL Server allows a user to manually override the maintenance of statistics in an individual database. The four main configurations controlling the automatic statistics maintenance behavior of SQL Server are as follows:

- New statistics on columns with no index (auto create statistics)
- Updating existing statistics (auto update statistics)
- The degree of sampling used to generate statistics
- Asynchronous updating of existing statistics (auto update statistics async)

You can control the preceding configurations at the levels of a database (all indexes and statistics on all tables) or on a case-by-case basis on individual indexes or statistics. The auto create statistics setting is applicable for nonindexed columns only because SQL Server always creates statistics for an index key when the index is created. The auto update statistics setting, and the asynchronous version, is applicable for statistics on both indexes and statistics on columns with no index.

Automatic Maintenance

By default, SQL Server automatically takes care of statistics. Both the auto create statistics and auto update statistics settings are on by default. As explained previously, it is usually better to keep these settings on. The auto update statistics async setting is off by default.

When you rebuild an index (if you choose to rebuild an index), it will create all new statistics for that index, based on a full scan of the data (more on that coming up). This means the rebuild process results in a very high-quality set of statistics, yet another way Microsoft helps you maintain your statistics automatically.

However, situations arise where creating and maintaining your statistics manually works better. For many of us, ensuring that our statistics are more up-to-date than the automated processes makes them means a higher degree of workload predictability. We know when and how we're maintaining the statistics because they are under our control. You also get to stop statistics maintenance from occurring randomly and control exactly when they occur, as well as control the recompiles that they lead to. This helps focus the load on your production system to nonpeak hours.

Auto Create Statistics

The auto create statistics feature automatically creates statistics on nonindexed columns when referred to in the `WHERE` clause of a query. For example, when this `SELECT` statement is run against the `Sales.SalesOrderHeader` table on a column with no index, statistics for the column are created:

```
SELECT  cc.CardNumber,
        cc.ExpMonth,
        cc.ExpYear
FROM    Sales.CreditCard AS cc
WHERE   cc.CardType = 'Vista';
```

Then the auto create statistics feature (make sure it is turned back on if you have turned it off) automatically creates statistics on column `CardType`. You can see this in the Extended Events session output in Figure 13-32.

name	batch_text	job_type	statistics_list
auto_stats	NULL	StatsUpdate	Created: CardType
auto_stats	NULL	StatsUpdate	Loading without updating: Sales.CreditCard._WA_Sys_000000...
auto_stats	NULL	StatsUpdate	Loading without updating: Sales.CreditCard.AK_CreditCard_Car...
sql_batch_completed	SELECT cc.CardNumber, cc.Ex...	NULL	NULL

Figure 13-32. *Session output with AUTO_CREATE_STATISTICS ON*

The `auto_stats` event fires to create the new set of statistics. You can see the details of what is happening in the `statistics_list` field `Created: CardType`. This is followed by the loading process of the new column statistic and a statistic on one of the indexes on the table and, finally, by the execution of the query.

Auto Update Statistics

The auto update statistics feature automatically updates existing statistics on the indexes and columns of a permanent table when the table is referred to in a query, provided the statistics have been marked as out-of-date. The types of changes are action statements, such as `INSERT`, `UPDATE`, and `DELETE`. The default threshold for the number of changes depends on the number of rows in the table. It's a fairly simple calculation.

```
Sqrt(1000*NumberOfRows)
```

This means if you had 500,000 rows in a table, then plugging that into the calculation results in 22,360.68. You would need to add, edit, or delete that many rows in your 500,000-row table before an automatic statistics update would occur.

For SQL Server 2014 and earlier, when not running under trace flag 2371, statistics are maintained as shown in Table 13-4.

Table 13-4. *Update Statistics Threshold for Number of Changes*

Number of Rows	Threshold for Number of Changes
0	> 1 insert
<500	> 500 changes
>500	20 percent of row changes

Row changes are counted as the number of inserts, updates, or deletes in the table.

Using a threshold reduces the frequency of the automatic update of statistics. For example, consider the following table:

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

CREATE TABLE dbo.Test1 (C1 INT);

CREATE INDEX ixl ON dbo.Test1 (C1);

INSERT INTO dbo.Test1 (C1)
VALUES (0);
```

After the nonclustered index is created, a single row is added to the table. This outdates the existing statistics on the nonclustered index. If the following `SELECT` statement is executed with a reference to the indexed column in the `WHERE` clause, like so, then the auto update statistics feature automatically updates statistics on the nonclustered index, as shown in the session output in Figure 13-33:

```
SELECT  C1
FROM    dbo.Test1
WHERE   C1 = 0;
```

Field	Value
async	False
attach_activity_id.g...	801C11DA-4063-4F21-A095-10655B72BB3A
attach_activity_id.s...	3
count	1
database_id	6
database_name	
duration	0
incremental	False
index_id	2
job_id	0
job_type	StatsUpdate
last_error	0
max_dop	-1
object_id	724197630
retries	0
sample_percentage	-1
statistics_list	Loading and updating: dbo.Test1.ix
status	Loading and updating stats
success	True

Figure 13-33. *Session output with AUTO_UPDATE_STATISTICS ON*

Once the statistics are updated, the change-tracking mechanisms for the corresponding tables are set to 0. This way, SQL Server keeps track of the number of changes to the tables and manages the frequency of automatic updates of statistics.

The new functionality of SQL Server 2016 and newer means that for larger tables, you will get more frequent statistics updates. You'll need to take advantage of trace flag 2371 on older versions of SQL Server to arrive at the same functionality. If automatic updates are not occurring frequently enough, you can take direct control, discussed in the "Manual Maintenance" section later in this chapter.