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;
```

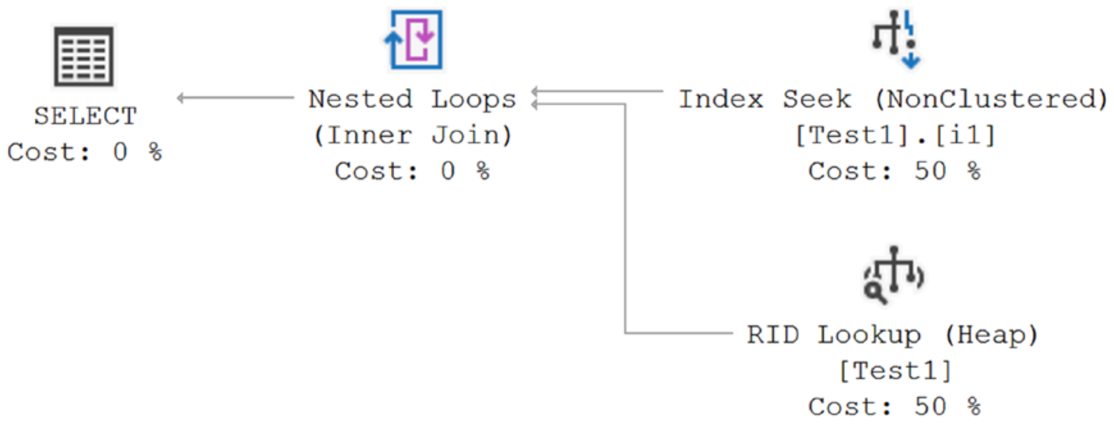


Figure 13-1. *Execution plan for a small result set*

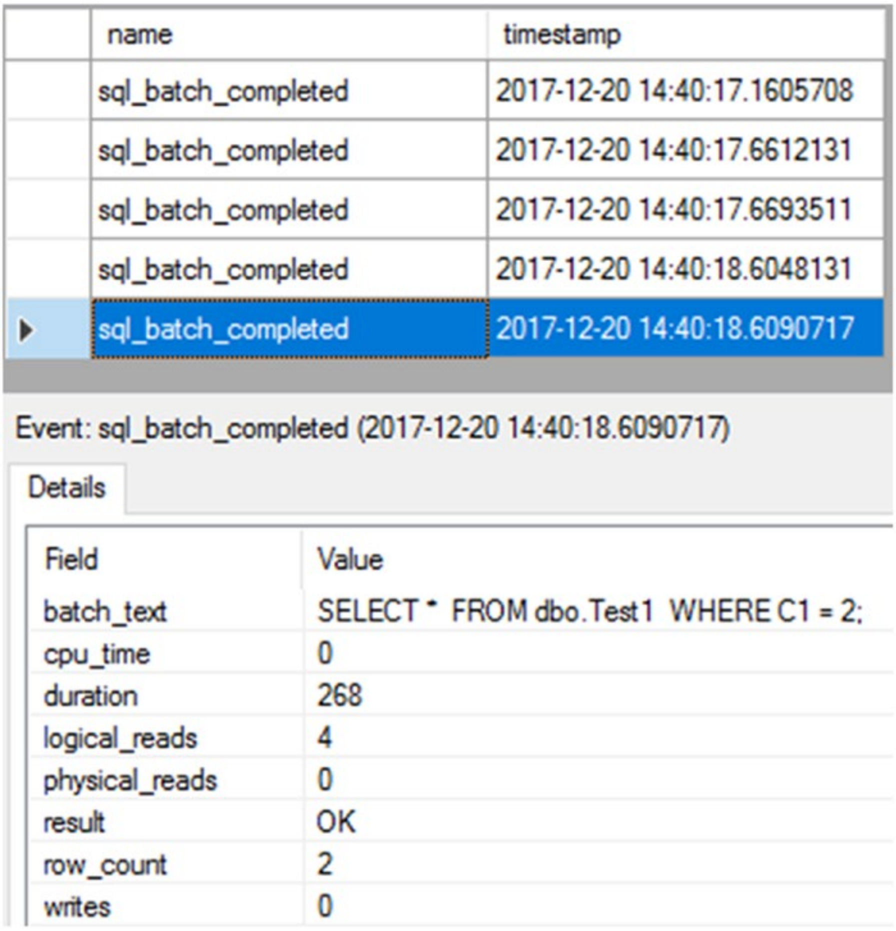
To understand the effect of small data modifications on a statistics update, create a session using Extended Events. In the session, add the event `auto_stats`, which captures statistics update and create events, and add `sql_batch_completed`. Here's the script to create and start an Extended Events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

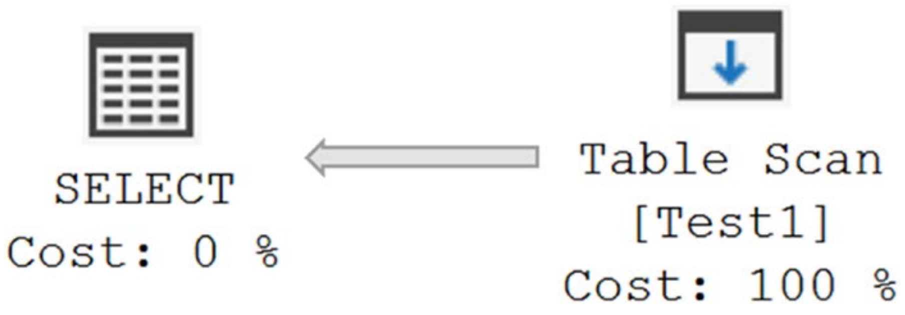
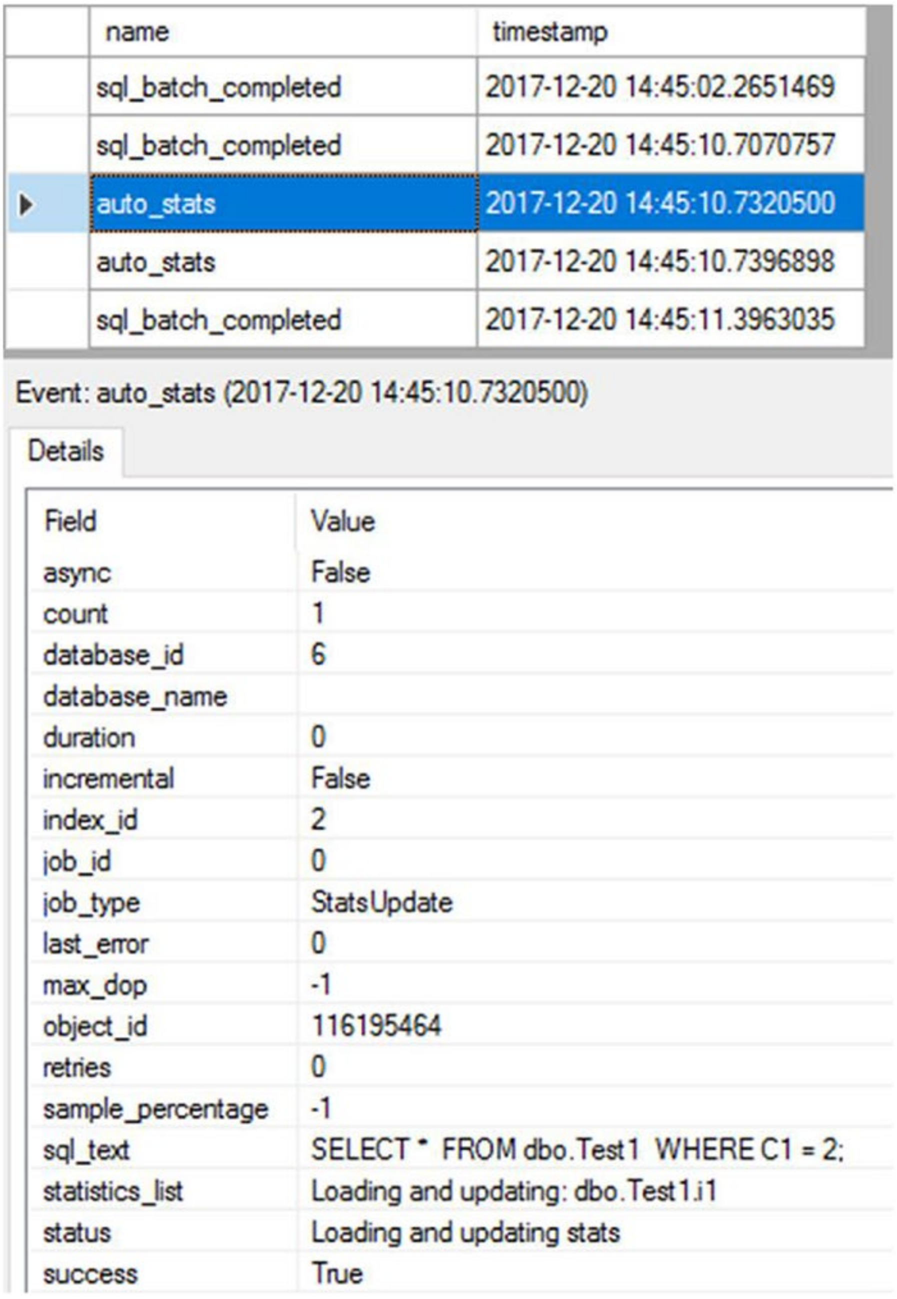


Figure 13-3. *Execution plan for a large result set*

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.

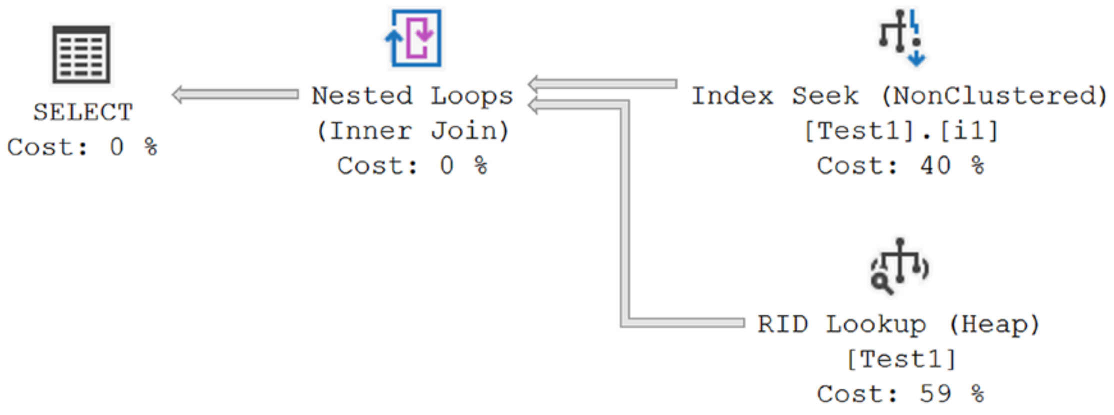
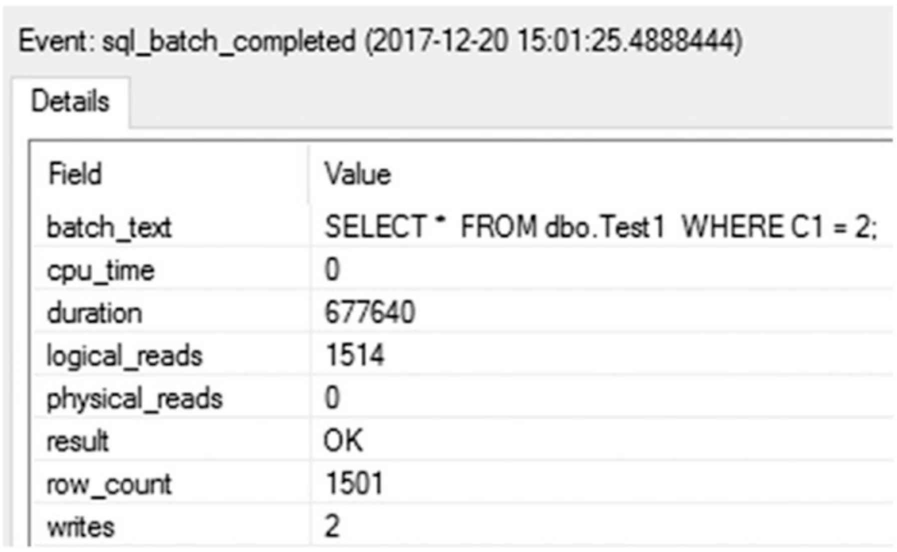


Figure 13-5. *Execution plan with AUTO_UPDATE_STATISTICS OFF*



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.

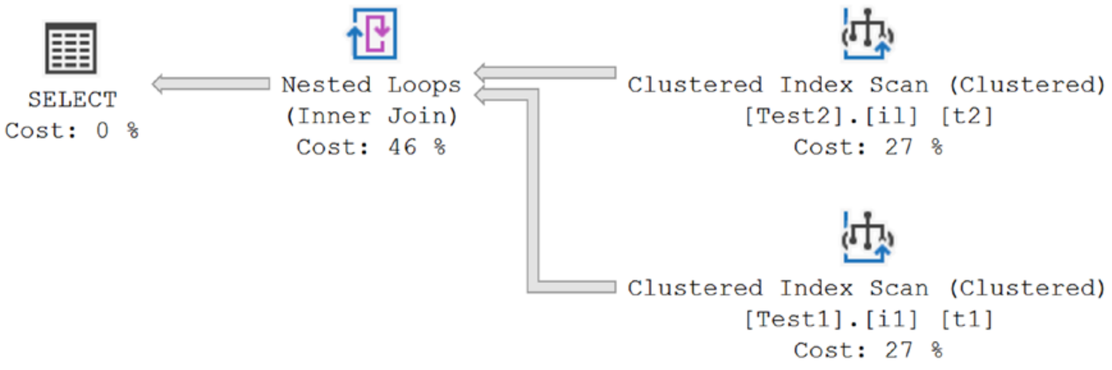
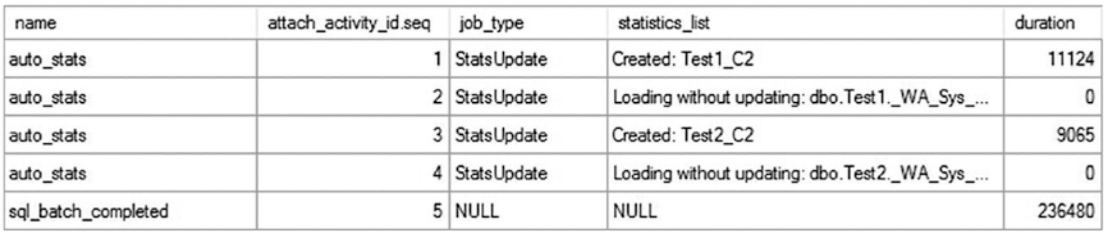


Figure 13-7. *Execution plan with AUTO_CREATE_STATISTICS ON*

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

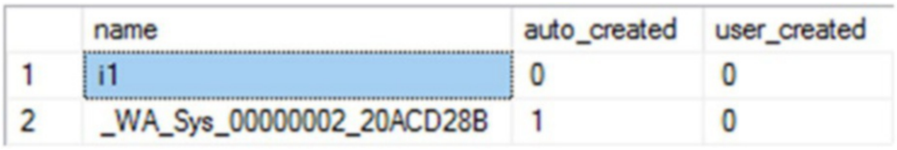
Figure 13-8. *Extended Events session output with AUTO_CREATE_STATISTICS ON*

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.

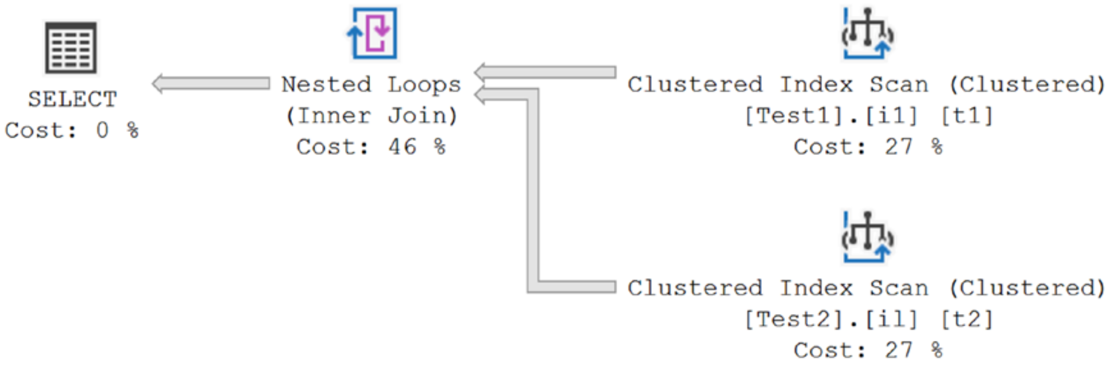
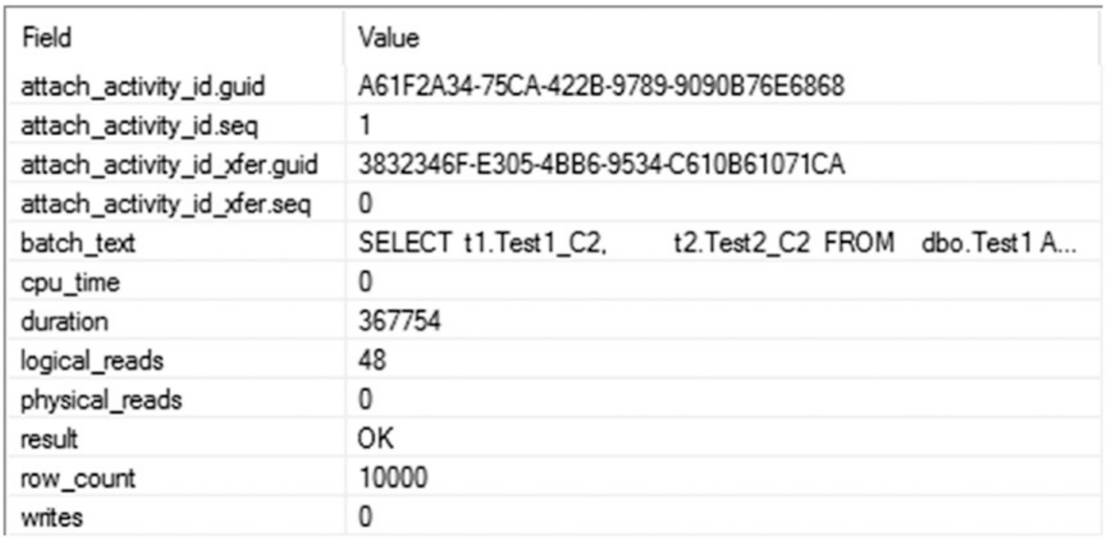


***Figure 13-10.** Execution plan for a different result set*



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.

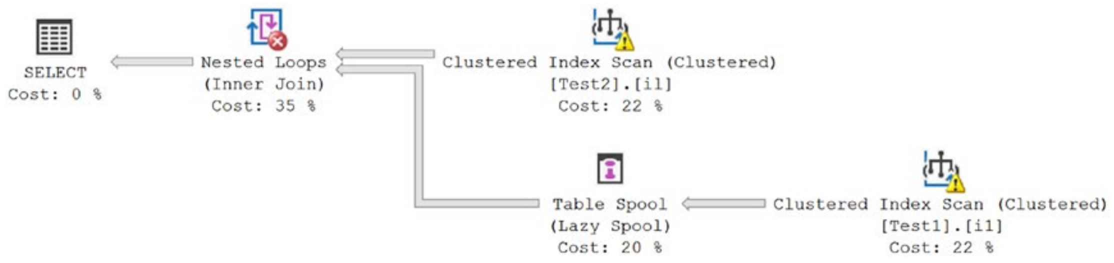
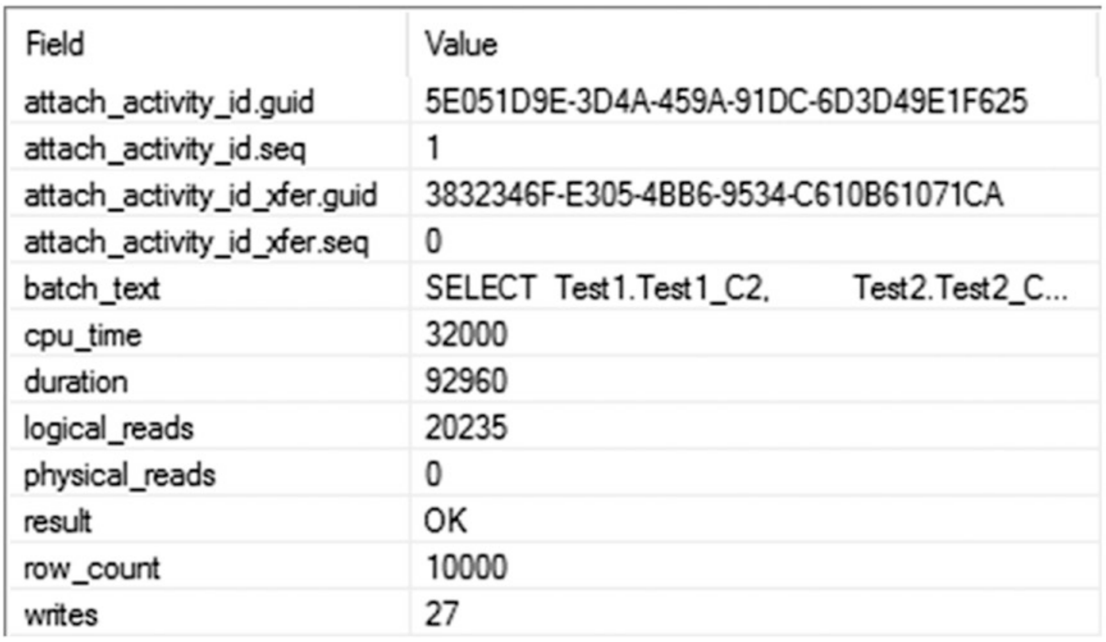


Figure 13-12. *Execution plan with AUTO_CREATE_STATISTICS OFF*



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,