**Statistics Used by the Query Optimizer in Microsoft SQL Server 2005**

By Author: Eric N. Hanson, Contributor: Lubor Kollar

Microsoft® SQL Server™ 2005 collects statistical information about indexes and column data stored in the database. These statistics are used by the SQL Server query optimizer to choose the most efficient plan for retrieving or updating data. This paper describes what data is collected, where it is stored, and which commands create, update, and delete statistics. By default, SQL Server 2005 also creates and updates statistics automatically, when such an operation is considered to be useful. This paper also outlines how these defaults can be changed on different levels (column, table, and database).

**On This Page**

[Statistical Data in SQLServer 2005](http://technet.microsoft.com/en-us/library/cc966419.aspx#XSLTsection123121120120)   
[Summary of Statistics Features](http://technet.microsoft.com/en-us/library/cc966419.aspx#XSLTsection124121120120)   
[What's New for Statistics in SQL Server 2005?](http://technet.microsoft.com/en-us/library/cc966419.aspx#XSLTsection125121120120)   
[Definitions](http://technet.microsoft.com/en-us/library/cc966419.aspx#XSLTsection126121120120)   
[Statistics Collected by SQL Server 2005](http://technet.microsoft.com/en-us/library/cc966419.aspx#XSLTsection127121120120)   
[Creating and Displaying Statistics: Example](http://technet.microsoft.com/en-us/library/cc966419.aspx#XSLTsection128121120120)   
[Creating Statistics with SQL Server 2005](http://technet.microsoft.com/en-us/library/cc966419.aspx#XSLTsection129121120120)   
[Maintaining Statistics in SQL Server 2005](http://technet.microsoft.com/en-us/library/cc966419.aspx#XSLTsection130121120120)   
[String Summary Statistics](http://technet.microsoft.com/en-us/library/cc966419.aspx#XSLTsection131121120120)   
[Statistics on Computed Columns](http://technet.microsoft.com/en-us/library/cc966419.aspx#XSLTsection132121120120)   
[Statistics on CLR User-Defined Type Columns](http://technet.microsoft.com/en-us/library/cc966419.aspx#XSLTsection133121120120)   
[Statistics and Indexed Views](http://technet.microsoft.com/en-us/library/cc966419.aspx#XSLTsection134121120120)   
[Best Practices for Managing Statistics](http://technet.microsoft.com/en-us/library/cc966419.aspx#XSLTsection135121120120)   
[Summary](http://technet.microsoft.com/en-us/library/cc966419.aspx#XSLTsection136121120120)   
[References](http://technet.microsoft.com/en-us/library/cc966419.aspx#XSLTsection137121120120)

**Statistical Data in SQLServer 2005**

Microsoft® SQL Server™ 2005 collects statistics about individual columns (single-column statistics) or sets of columns (multi-column statistics). Statistics are used by the query optimizer to estimate the selectivity of expressions, and thus the size of intermediate and final query results. Good statistics allow the optimizer to accurately assess the cost of different query plans, and choose a high-quality plan. All information about a single statistics object is stored in several columns of a single row in the **sysindexes** table, and in a statistics binary large object (**statblob**) kept in an internal-only table. In addition, information about statistics can be found in the new metadata views **sys.stats** and **sys.indexes**.

[Top Of Page](http://technet.microsoft.com/en-us/library/cc966419.aspx#mainSection)

**Summary of Statistics Features**

SQL Server 2005 has a number of features for maintaining statistics. The most important one is the ability to automatically create and update statistics. This feature is on by default in SQL Server 2005 and SQL Server 2000. Approximately 98% of SQL Server 2000 installations leave this feature enabled, which is a best practice. For the majority of databases and applications, developers and administrators can rely on the automatic creation and update of statistics to provide comprehensive and accurate statistics about their data. This allows the SQL Server 2005 query optimizer to produce good query plans consistently, while keeping development and administration costs low. If you need more control over statistics creation and update to get good query plans and manage the overhead of gathering statistics, you can use manual statistics creation and update capabilities.

An important new feature for high-throughput online transaction processing application environments is the ability to asynchronously update statistics automatically. This can improve predictability of query response time in such environments.

SQL Server 2005 statistics features allow you to:

* **implicitly create and update statistics** with the default sampling rate (in the SELECT, INSERT, DELETE, and UPDATE commands, use of a column in a query condition such as a WHERE or JOIN clause causes statistics to be created or updated if necessary when automatic statistics creation and update is enabled)
* **manually create and update statistics** with any desired sampling rate, and drop statistics (CREATE STATISTICS, UPDATE STATISTICS, DROP STATISTICS, CREATE INDEX, DROP INDEX)
* **manually create statistics in bulk** for all columns of all tables in a database (**sp\_createstats**)
* **manually update all existing statistics** in the database (**sp\_updatestats**)
* **list statistics objects** that exist for a table or database (**sp\_helpstats**, catalog views **sys.stats, sys.stats\_columns**)
* **display descriptive information about statisticsobjects** (DBCC SHOW\_STATISTICS)
* **enable and disable automatic creation and update of statistics** database-wide or for a specific table or statistics object (ALTER DATABASE options AUTO\_CREATE\_STATISTICS and AUTO\_UPDATE\_STATISTICS, sp\_autostats, and the NORECOMPUTE option on CREATE STATISTICS and UPDATE STATISTICS)
* **enable and disable asynchronous automatic update ofstatistics** (ALTER DATABASE option AUTO\_UPDATE\_STATISTICS\_ASYNC)

In addition, SQL Server Management Studio allows you to graphically browse and control statistics objects within its Object Explorer view. Statistics are listed in Object Explorer in a folder underneath each table object.

[Top Of Page](http://technet.microsoft.com/en-us/library/cc966419.aspx#mainSection)

**What's New for Statistics in SQL Server 2005?**

SQL Server 2005 has a number of new statistics features that allow the query optimizer to choose better query plans for a broader range of queries, or otherwise improve the management of statistics. These include the following enhancements:

* **String summary statistics**: Information about the frequency distribution of substrings is maintained for character columns. This helps the optimizer better estimate the selectivity of conditions that use the LIKE operator.
* **Asynchronous auto update statistics**: The ALTER DATABASE option AUTO\_UPDATE\_STATISTICS\_ASYNC is new in SQL Server 2005. The option is off by default. When enabled, this option causes SQL Server 2005 to perform automatic update of statistics in the background. The query that caused the statistics update does not block. It proceeds with the old statistics. This provides more predictable query response time for some work loads.
* **Computed column statistics**: Statistics can be manually or automatically created and updated on computed columns (this was partially supported in SQL Server 2000 but not documented as such).
* **Large object support**: Columns of type **ntext**, **text**, and **image**, as well as the new types **nvarchar(max)**, **varchar(max)**, and **varbinary(max)** can be specified as statistics columns.
* **Improved statistics loading framework**: The optimizer internally improves the loading of statistics compared with SQL Server 2000. It now loads all and only the statistics it needs, improving optimization result quality and performance.
* **Increased ability to automatically create statistics on computed columns**: If a query contains an expression equivalent to the expression of a computed column, SQL Server 2005 can auto create statistics on the computed column if they are needed.
* **Minimum sample size**: A minimum of 8 megabytes of data, or the size of the table if smaller, are now sampled during statistics gathering.
* **Increased limit on number of statistics**: The number of column statistics objects allowed per table has been increased to 2,000. An additional 249 index statistics may also be present, for a maximum of 2,249 statistics per table.
* **Enhanced DBCC SHOW\_STATISTICS output**: DBCC SHOW\_STATISTICS now displays the name of the statistics object being displayed, to avoid ambiguity.
* **Statistics auto update is now based on column modification counters**: In SQL Server 2000, statistics update was determined by the number of row changes. Now, changes are tracked at the column level, and auto update of statistics can now be avoided on columns which have not changed enough to warrant statistics update.
* **Statistics on internal tables**: Statistics are fully supported on the tables listed in **sys.internal\_tables**, including XML and fulltext indexes, service broker queues, and query notification tables.
* **Single rowset output for DBCC SHOW\_STATISTICS**: DBCC SHOW\_STATISTICS supports options to output the header, density vector, and histogram individually as single rowsets. This allows easier programming when automatically processing DBCC SHOW\_STATISTICS output.
* **Statistics on up-to 32 columns**: The limit on the number of columns in a statistics object has been increased to 32 from 16.
* **Statistics on partitioned tables**: Statistics are fully supported on partitioned tables, which are new in SQL Server 2005. Histograms are maintained on a per-table basis (not per-partition).
* **Parallel statistics gathering for fullscan**: For statistics gathered with **fullscan**, creation of a single statistics object can be done in parallel for both non-partitioned and partitioned tables.
* **Improved recompiles and statistics creation in case of missing statistics**: In the case where auto create of statistics failed, on a subsequent execution of a plan generated with missing statistics, auto create is performed and the plan is recompiled; the missing statistics condition does not persist. See the [Batch Compilation, Recompilation, and Plan Caching Issues in SQL Server 2005](http://technet.microsoft.com/en-us/library/cc966425.aspx) white paper by Marathe for more details.
* **Improved recompilation logic and statistics update for empty tables**: Changing from 0 to > 0 rows in a table results in query recompilation and update of statistics. See the [Batch Compilation, Recompilation, and Plan Caching Issues in SQL Server 2005](http://technet.microsoft.com/en-us/library/cc966425.aspx) white paper by Marathe for more details.
* **Clearer and more consistent display of histograms**: DBCC SHOW\_STATISTICS is improved because histograms are always scaled before being stored in the catalogs.
* **Inferred date correlation constraints**: By enabling the DATE\_CORRELATION\_OPTIMIZATION database setting, you can cause SQL Server to maintain information about the correlation between datetime fields across a pair of tables linked by a foreign key. This information is used to allow implied predicates to be determined for some queries. The information is not used directly for selectivity estimation or costing by the optimizer, so it is not statistics in the strictest sense, but it is closely related to statistics because it is auxiliary information used to help obtain a better query plan.
* **sp\_updatestats**: In Microsoft SQL Server 2005, **sp\_updatestats** updates only those statistics that require updating based on the **rowmodctr** information in the **sys.sysindexes** compatibility view, thus eliminating unnecessary updates of unchanged items. For databases running under compatibility level 90 or higher, **sp\_updatestats** preserves the automatic UPDATE STATISTICS setting for any particular index or statistics.

There are also some other minor changes in behavior for statistics. In particular, the **statblob** column in **sys.sysindexes** is now always given as NULL. The **statblob** itself is kept in an internal-only catalog table.

[Top Of Page](http://technet.microsoft.com/en-us/library/cc966419.aspx#mainSection)

**Definitions**

We define the following terms related to SQL Server 2005 statistics:

* **statblob**: Statistics Binary Large Object. This object is stored in the system catalogs in an internal catalog view, **sys.sysobjvalues**.
* **String Summary**: A form of statistics that summarizes the frequency distribution of substrings in a string column. This is used to help estimate selectivity of LIKE predicates. It is stored in the **statblob** for a string column.
* **sysindexes**: The **sys.sysindexes** catalog view, which contains information about tables and indexes.
* **Predicate**: A condition that evaluates to true or false. Predicates appear in a WHERE or JOIN clause in a database query.
* **Selectivity**: The fraction of rows from the input set of the predicate that satisfy the predicate. More sophisticated selectivity measures are also used to estimate the number of rows produced by joins, DISTINCT, and other operators. For example, SQL Server 2005 estimates the selectivity of the predicate "Sales.SalesOrderHeader.OrderID = 43659" in the AdventureWorks database as 1/31465 = 0.00003178.
* **Cardinality estimate**: An estimate of the size of a result set. For example, if a table T has 100,000 rows and a query contains a selection predicate of the form T.a=10, and a histogram shows that the selectivity of T.a=10 is 10%, then the cardinality estimate for the fraction of rows of T that must be considered by the query is 10% \* 100,000 = 10,000.
* **LOB**: Large object (value of type **text**, **ntext**, **image**, **varchar(max)**, **nvarchar(max)**, **varbinary(max)**).

[Top Of Page](http://technet.microsoft.com/en-us/library/cc966419.aspx#mainSection)

**Statistics Collected by SQL Server 2005**

SQL Server 2005 maintains the following information at the table level. These are not part of a statistics object, but SQL Server 2005 uses them in some cases during query cost estimation.

* Number of rows in the table or index (**rows** column in **sys.sysindexes**).
* Number of pages occupied by the table or index (**dpages** column in **sys.sysindexes**).

SQL Server 2005 collects the following statistics about table columns and stores them in a statistics object (**statblob**):

* Time the statistics were collected.
* Number of rows used to produce the histogram and density information (described below).
* Average key length.
* Single-column histogram, including the number of steps.
* A string summary, if the column contains character data. DBCC SHOW\_STATISTICS output contains a column "String Index" which has the value YES if a statistics object contains a string summary.

A histogram is a set of up to 200 values of a given column. All or a sample of the values in a given column are sorted; the ordered sequence is divided into up to 199 intervals so that the most statistically significant information is captured. In general, these intervals are of nonequal size. The following values, or information sufficient to derive them, are stored with each step of the histogram.

|  |  |
| --- | --- |
| RANGE\_HI\_KEY | A key value showing the upper boundary of a histogram step. |
| RANGE\_ROWS | Specifies how many rows are inside the range (they are smaller than this RANGE\_HI\_KEY, but bigger than the previous smaller RANGE\_HI\_KEY). |
| EQ\_ROWS | Specifies how many rows are exactly equal to RANGE\_HI\_KEY. |
| AVG\_RANGE\_ROWS | Average number of rows per distinct value inside the range. |
| DISTINCT\_RANGE\_ROWS | Specifies how many distinct key values are inside this range (not including the previous key before RANGE\_HI\_KEY and RANGE\_HI\_KEY itself); |

Histograms in SQL Server 2005 are only built for a single column—the first column in the set of key columns of the statistics object.

SQL Server 2005 builds the histogram from the sorted set of column values in three steps:

* **Histogram initialization**: In the first step, a sequence of values starting at the beginning of the sorted set is processed, and up to 200 values of RANGE\_HI\_KEY, EQ\_ROWS, RANGE\_ROWS, and DISTINCT\_RANGE\_ROWS are collected (RANGE\_ROWS and DISTINCT\_RANGE\_ROWS are always zero during this step). The first step ends either when all input has been exhausted, or when 200 values have been found.
* **Scan with bucket merge**: Each additional value from the leading column of the statistics key is processed in the second step, in sorted order; each successive value is either added to the last range or a new range at the end is created (this is possible because the input values are sorted). If a new range is created, then one pair of existing, neighboring ranges is collapsed into a single range. This pair of ranges is selected in order to minimize information loss. The number of steps after collapsing ranges stays at 200 throughout this step. This method is based on a variation of the *maxdiff* histogram.
* **Histogram consolidation**: In the third step, more ranges may be collapsed if a significant amount of information is not lost. Therefore, even if the column has more than 200 unique values, the histogram may have less than 200 steps.

If the histogram has been built using a sample, then the values of RANGE\_ROWS, EQ\_ROWS, DISTINCT\_RANGE\_ROWS and AVG\_RANGE\_ROWS are estimated and therefore they do not need to be whole integers.

Density is information about the number of duplicates in a given column or combination of columns and it is calculated as 1/(number of distinct values). When a column is used in an equality predicate then the number of qualifying rows is estimated using the density derived from the histogram. Histograms are also used to estimate the selectivity of non-equality selection predicates, joins, and other operators.

In addition to the timestamp showing the time the statistics were gathered, the number of rows in the table, the number of rows sampled to produce the histogram, the Density information and average key length, and the histogram itself, single-column statistics information includes an **All density** value for each set of columns forming a prefix of the statistics column set. This is shown in the second rowset output by DBCC SHOW\_STATISTICS. **All density** is an estimate of 1/(number of distinct values in the prefix column set). The next section gives an example of this.

**Note**: The Density value included in the first row returned by **dbcc show\_statistics** is the density of all values sampled other than the RANGE\_HI\_KEY values. The RANGE\_HI\_KEY values are typically the more frequent values in the distribution. Hence, the displayed Density gives potentially useful information about the density of non-frequent values.

The multi-column statistics for one set of columns consists of one histogram for the first column in the statistics definition, one density value for the first column, and an All Density value for each prefix combination of columns (including the first column alone). Each set of multi-column statistics (a histogram and two or more density values) is stored in one **statblob** together with the timestamp of the last statistics update, the number of rows in the sample used to produce the statistical information, the number of steps in the histogram, and the average length of the key. A string summary is included for only the first column, only if it contains character data.

Use **sp\_helpindex** and **sp\_helpstats** to display the list of all statistics available for a given table. **sp\_helpindex** lists all indexes on the table, and **sp\_helpstats** lists all the statistics on the table. Each index also carries the statistical information for its columns. The statistical information created using the CREATE STATISTICS command is equivalent to the statistics built by a CREATE INDEX command on the same columns. The only difference is that the CREATE STATISTICS command uses sampling by default while the CREATE INDEX command gathers the statistics with fullscan since it has to process all rows for the index anyway.

[Top Of Page](http://technet.microsoft.com/en-us/library/cc966419.aspx#mainSection)

**Creating and Displaying Statistics: Example**

The following example illustrates how statistics are created both automatically and manually, and shows how to list and display information about statistics. Results are given for some but not all commands; the output produced by SQL Server 2005 is shown when this is useful to illustrate its behavior. You can run this example yourself to see the complete set of output.

USE tempdb

GO

-- Clean up objects from any previous runs.

IF object\_id(N'Person.Contact','U') IS NOT NULL

DROP TABLE Person.Contact

GO

IF EXISTS (SELECT \* FROM sys.schemas WHERE name = N'Person')

DROP SCHEMA Person

GO

-- Create a sample schema and table.

CREATE SCHEMA Person

GO

CREATE TABLE Person.Contact(

FirstName nvarchar(60),

LastName nvarchar(60),

Phone nvarchar(15),

Title nvarchar(15)

)

GO

-- Populate the table with a few rows.

INSERT INTO Person.Contact

VALUES(N'James',N'Smith',N'425-555-1234',N'Mr')

INSERT INTO Person.Contact

VALUES(N'James',N'Andersen',N'425-555-1111',N'Mr')

INSERT INTO Person.Contact

VALUES(N'James',N'Andersen',N'425-555-3333',N'Mr')

INSERT INTO Person.Contact

VALUES(N'Christine',N'Williams',N'425-555-0000',N'Dr')

INSERT INTO Person.Contact

VALUES(N'Susan',N'Zhang',N'425-555-2222',N'Ms')

GO

-- Show that there are no statistics yet on the Person.Contact table.

sp\_helpstats N'Person.Contact', 'ALL'

GO

-- Implicitly create statistics on LastName.

SELECT \* FROM Person.Contact WHERE LastName = N'Andersen'

GO

-- Show that statistics were automatically created on LastName.

sp\_helpstats N'Person.Contact', 'ALL'

GO

**Results**:

|  |  |
| --- | --- |
| **statistics\_name** | **statistics\_keys** |
| \_WA\_Sys\_00000002\_1B29035F | LastName |

-- Create an index, which also creates statistics.

CREATE NONCLUSTERED INDEX Phone on Person.Contact(Phone)

GO

-- Show that creating the index created an associated statistics object.

sp\_helpstats N'Person.Contact', 'ALL'

GO

**Results**:

|  |  |
| --- | --- |
| statistics\_name | statistics\_keys |
| \_WA\_Sys\_00000002\_1B29035F | LastName |
| Phone | Phone |

-- Create a multi-column statistics object on first and last name.

CREATE STATISTICS FirstLast ON Person.Contact(FirstName,LastName)

GO

-- Show that there are now three statistics objects on the table.

sp\_helpstats N'Person.Contact', 'ALL'

GO

**Results**:

|  |  |
| --- | --- |
| statistics\_name | statistics\_keys |
| \_WA\_Sys\_00000002\_1B29035F | LastName |
| FirstLast | FirstName, LastName |
| Phone | Phone |

-- Display the statistics for LastName.

DBCC SHOW\_STATISTICS (N'Person.Contact', LastName)

GO

**Results**:

**Statistics header information**:

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| **Name** | **Updated** | **Rows** | **Rows Sampled** | **Steps** | **Density** | **Average key length** | **String Index** |
| \_WA\_Sys\_00000002\_1B29035F | Mar 25 2005 11:21AM | 5 | 5 | 4 | 0 | 13.6 | YES |

**Column set prefixes and associated densities and lengths**:

|  |  |  |
| --- | --- | --- |
| **All Density** | **Average Length** | **Columns** |
| 0.25 | 13.6 | LastName |

**Histogram steps**:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **RANGE\_HI\_KEY** | **RANGE\_ROWS** | **EQ\_ROWS** | **DISTINCT\_RANGE\_ROWS** | **AVG\_RANGE\_ROWS** |
| Andersen | 0 | 2 | 0 | 0 |
| Smith | 0 | 1 | 0 | 1 |
| Williams | 0 | 1 | 0 | 1 |
| Zhang | 0 | 1 | 0 | 1 |

-- If you take the name of the statistics object displayed by

-- the command above and subsitute it in as the second argument of

-- DBCC SHOW\_STATISTICS you can form a command like the following one

--(the exact name of the automatically created statistics object

-- will typically be different for you).

DBCC SHOW\_STATISTICS (N'Person.Contact', \_WA\_Sys\_00000002\_2D7CBDC4)

-- Executing the above command illustrates that you can show statistics by

-- column name or statistics object name.

GO

-- The following displays multi-column statistics. Notice the two

-- different density groups for the second rowset in the output.

DBCC SHOW\_STATISTICS (N'Person.Contact', FirstLast)

**Results (second rowset only)**:

**Column set prefixes and associated densities and lengths**:

|  |  |  |
| --- | --- | --- |
| **All density** | **Average Length** | **Columns** |
| 0.3333333 | 11.6 | FirstName |
| 0.25 | 25.2 | FirstName, LastName |

To see a fully populated histogram for a larger table, run the following commands:

USE AdventureWorks

-- Clean up objects from previous runs.

IF EXISTS (SELECT \* FROM sys.stats

WHERE object\_id = object\_id('Sales.SalesOrderHeader')

AND name = 'TotalDue')

DROP STATISTICS Sales.SalesOrderHeader.TotalDue

GO

CREATE STATISTICS TotalDue ON Sales.SalesOrderHeader(TotalDue)

GO

DBCC SHOW\_STATISTICS(N'Sales.SalesOrderHeader', TotalDue)

[Top Of Page](http://technet.microsoft.com/en-us/library/cc966419.aspx#mainSection)

**Creating Statistics with SQL Server 2005**

You can create statistics in SQL Server 2005 in a number of different ways, as described below.

* The optimizer automatically creates single-column statistics as needed as a side effect of optimizing SELECT, INSERT, UPDATE, and DELETE statements when AUTO\_CREATE\_STATISTICS is enabled, which is the default setting.
* There are two basic statements in SQL Server 2005 that explicitly generate the statistical information described above: CREATE INDEX generates the declared index in the first place, and also creates one set of statistics for the column combinations constituting the index keys (but not other included columns). CREATE STATISTICS only generates the statistics for a given column or combination of columns.
* In addition, there are several other ways to create statistics or indices. Ultimately, though, each issues one of the above two commands. Use **sp\_createstats** to create statistics for all eligible columns (all except XML columns) for all user tables in the current database. A new statistics object will not be created for columns that already have a statistics object.
* Use **dbcc dbreindex** to rebuild one or more indexes for a table in the specified database.
* In Management Studio, expand the folder under a Table object, right click the **Statistics** folder, and choose **New Statistics**.
* Use the Database Tuning Advisor (DTA) to create indexes.

Here is an example of a CREATE STATISTICS command on the **AdventureWorks.Person.Contact** table:

CREATE STATISTICS FirstLast2 ON Person.Contact(FirstName,LastName)

WITH SAMPLE 50 PERCENT

Usually, statistics with default sampling are sufficient to allow good query plans to be produced. However, there may be cases when statistics with larger sample sizes may benefit query optimization, such as when the values in the given column sample are not random. Non-random samples may occur if the data is sorted or clustered. Sorting or clustering may be caused by the creation of indexes, or loading data into a heap structure that is already sorted or clustered. The most commonly used larger sample size is **fullscan** because it gives the most accurate statistics. The price for using statistics with larger sample sizes is the time required to create the statistics.

The above command creates a single two-column statistics object. In this case, the SAMPLE 50 PERCENT is ignored and a full scan is performed because the table is too small. Sampling is used primarily to avoid excessive scans of data and affects only tables and indices with 1,024 or more pages (8 MB).

In SQL Server 2005, statistics are created for all indices at the index creation time. SQL Server creates single-column statistics automatically when compiling queries. These statistics are created for columns where the optimizer would otherwise have to estimate the approximate density or distribution. There are the following exceptions to this rule. Statistics may not be created automatically when (1) the database is read-only, (2) there are too many outstanding compilations in progress, and (3) the column's data type is not supported for auto stats.

The automatic statistics creation function can be disabled at the database level by executing

ALTER DATABASE dbname SET AUTO\_CREATE\_STATISTICS OFF

Similarly, automatic statistics creation can be enabled at the database level by executing

ALTER DATABASE dbname SET AUTO\_CREATE\_STATISTICS ON

It's recommended that you leave this setting ON. Only disable it if you need to do so to work around known performance issues, such as the need to specify other than the default sampling rate for some tables.

By default, statistics are created by sampling the data set when executing the CREATE STATISTICS command or when the statistics are automatically created. CREATE INDEX scans the whole data set anyway; therefore, the index statistics are initially created without sampling (the equivalent of **fullscan**). The CREATE STATISTICS command allows you to set the sample size in the WITH clause either by specifying **fullscan** or the percentage of data or number of rows to scan. The latter is interpreted as an approximation. It is also possible to inherit the previous sample size when specifying WITH RESAMPLE on the UPDATE STATISTICS command. This is particularly useful when there are indexes on some columns or column sets (originally created with **fullscan** statistics) and only statistics on some other columns or column sets (originally created with SAMPLE statistics). Using the **resample** option on UPDATE STATISTICS will then maintain the **fullscan** statistics for the indexes and sample statistics for the rest of the columns.

For small tables, a minimum of 8 MB of data is sampled. If a table starts small, and you sample with the default sampling rate, and thereafter use the **resample** option when updating the statistics, you will get the equivalent of **fullscan** even as the table grows beyond 8 MB. Avoid the use of **resample** if you want the default sampling rate as table size varies.

The **resample** sampling rate is computed as a function of the number of rows sampled and the total number of rows in the table during the previous statistics calculation. Since the actual sampling rate may vary due to the random nature of sampling, the **resample** rate is only an approximation of the prior sampling rate for non-**fullscan** samples. For consistently repeatable sampling, explicitly specify the same sampling rate you used before with UPDATE STATISTICS instead of using **resample**.

The **dbcc show\_statistics** command displays the sample size under the Rows Sampled heading. Statistics created automatically, or updated automatically (as described in the next section) are always generated using default sampling. The default sampling rate is a slow-growing function of the table size, which allows statistics to be gathered relatively quickly even for very large tables.

When statistics are created and updated, the optimizer must choose an access path to gather the statistics. An access path may include a heap, clustered index, or non-clustered index. For sampled statistics, the optimizer tries to avoid access paths that are physically sorted on the leading statistics key column. This helps provide a more random sample, which leads to more accurate statistics. For those access paths that are not sorted on the statistics key (if such an access path exists) the lowest cost access path is chosen. This is the narrowest index or the heap. For **fullscan** statistics, the sort order of the access paths is not significant for statistical accuracy, so the lowest cost access path is chosen.

SQL Server Profiler can monitor automatic statistics creation. The Auto Stats event is in the group of Performance trace events. Select also the IntegerData, Success and ObjectID columns for the Auto Stats event when defining the trace. Once the AutoStats event is captured, the Integer Data column contains the number of statistics created or updated for a given table, the Object ID is the ID of the table and the TextData column (included in the trace definition by default) contains names of the columns for which statistics creation or update was performed, together with either an **Updated**: or **Created**: prefix. The Success column contains an indication of the success or failure of the Auto Stats operation. Specifically, Success has three possible values:

|  |  |  |
| --- | --- | --- |
| **Name** | **Value** | **Definition** |
| FAILED | 0 | Auto Stats creation or update failed for reasons other than THROTTLED (see below). For example, the database was read-only. |
| SUCCESS | 1 | Auto Stats creation or update succeeded |
| THROTTLED | 2 | Auto Stats creation or update failed because too many optimizations were in progress. |

In some cases, you may also observe an AutoStats event in which no statistics are created or updated. Such an event is generated when **auto update statistics** is turned off, or when there have been a substantial number of changes to a table referenced in a query, and the optimizer has been able to remove all references to that table from the query plan because of the structure of the query and the existence of a foreign key constraint.

The DROP STATISTICS command is used to drop statistics, but it is not possible to drop statistics that are a byproduct of an index. Such statistics are removed only when the index is dropped.

[Top Of Page](http://technet.microsoft.com/en-us/library/cc966419.aspx#mainSection)

**Maintaining Statistics in SQL Server 2005**

After a series of INSERTs, DELETEs, and/or UPDATEs are performed on a table, the statistics may not reflect the true data distribution in a given column or index. If the SQL Server query optimizer requires statistics for a particular column in a table that has undergone substantial update activity since the last time the statistics were created or updated, SQL Server automatically updates the statistics by sampling the column values (using **auto update statistics**). The statistics auto update is triggered by query optimization or by execution of a compiled plan, and involves only a subset of the columns referred to in the query. Statistics are updated before query compilation if AUTO\_UPDATE\_STATISTCS\_ASYNC is OFF, and asynchronously if it is ON.

When a query is first compiled, if the optimizer needs a particular statistics object, and that statistics object exists, the statistics object is updated if it is out of date. When a query is executed and its plan is in the cache, the statistics the plan depends on are checked to see if they are out of date. If so, the plan is removed from the cache and during re-compilation of the query the statistics are updated. The plan also is removed from the cache if any of the statistics it depends on have changed.

SQL Server 2005 determines whether to update statistics based on changes to *column modification counters* (**colmodctrs**).

A statistics object is considered out of date in the following cases:

* If the statistics is defined on a regular table, it is out of date if:
  1. The table size has gone from 0 to >0 rows.
  2. The number of rows in the table when the statistics were gathered was 500 or less, and the colmodctr of the leading column of the statistics object has changed by more than 500 since then.
  3. The table had more than 500 rows when the statistics were gathered, and the colmodctr of the leading column of the statistics object has changed by more than 500 + 20% of the number of rows in the table when the statistics were gathered.
* If the statistics object is defined on a temporary table, it is out of date as discussed above, except that there is an additional threshold for recomputation at 6 rows, with a test otherwise identical to test 2 in the previous list.

Table variables do not have statistics at all.

The **auto update statistics** feature described above may be turned off at different levels.

* On the database level, disable auto update statistics using:
* ALTER DATABASE dbname SET AUTO\_UPDATE\_STATISTICS OFF
* At the table level, disable auto update statistics using the NORECOMPUTE option of the UPDATE STATISTICS command or CREATE STATISTICS command.
* Use **sp\_autostats** to display and change the auto update statistics setting for a table, index, or statistics object.

Re-enabling the automatic updating of statistics can be done similarly using ALTER DATABASE, UPDATE STATISTICS, or **sp\_autostats**.

SQL Server 2005 maintains the automatic statistics update setting on per-database, per-table, and per-index or statistics object levels. While one can change this setting for all statistics on one table using a single **sp\_autostats** command, this is implemented by changing the setting individually for each statistics object and index on a given table. No metadata explicitly records that auto update statistics is ON or OFF for the whole table.

The following table shows the combined effect of different database, and table and index settings.

|  |  |  |
| --- | --- | --- |
| **Database Setting** | **Index or Statistics Object Setting** | **Statistics Auto Update is in Effect for the Object** |
| ON | ON | ON |
| ON | OFF | OFF |
| OFF | ON | OFF |
| OFF | OFF | OFF |

It is not possible to override the database setting of OFF for auto update statistics by setting it ON at the statistics object level.

Auto statistics update is always performed by sampling the index or table using the default sampling rate. To set the sampling rate explicitly, run CREATE or UPDATE STATISTICS

Statistics update is covered by the same SQL Profiler event as statistics creation.

[Top Of Page](http://technet.microsoft.com/en-us/library/cc966419.aspx#mainSection)

**String Summary Statistics**

SQL Server 2005 includes patented technology for estimating the selectivity of LIKE conditions. It builds a statistical summary of substring frequency distribution for character columns (a string summary). This includes columns of type **text**, **ntext**, **char**, **varchar**, and **nvarchar**. Using the string summary, SQL Server can accurately estimate the selectivity of LIKE conditions where the pattern may have any number of wildcards in any combination. For example, SQL Server can estimate the selectivity of predicates of the following form:

Column LIKE 'string%'

Column LIKE '%string'

Column LIKE '%string%'

Column LIKE 'string'

Column LIKE 'str\_ing'

Column LIKE 'str[abc]ing'

Column LIKE '%abc%xy'

If there is a user-specified escape character in a LIKE pattern (i.e., the pattern is of the form LIKE *pattern* ESCAPE *escape\_character*), then SQL Server 2005 guesses selectivity.

This is an improvement over SQL Server 2000, which uses a guess for selectivity when any wildcard other than a trailing wildcard % is used in the LIKE pattern, and has limited accuracy in its estimates in that case.

The String Index field in the first row set returned by DBCC SHOW\_STATISTICS includes the value YES if the statistics object also includes a string summary. The contents of the string summary are not shown. The string summary includes additional information beyond what is shown in the histogram.

For strings longer than 80 characters, the first and last 40 characters are extracted from the string and concatenated prior to considering the string in the creation of the string summary. Hence, accurate frequency estimates for substrings that appear only in the ignored portion of a string are not available.

[Top Of Page](http://technet.microsoft.com/en-us/library/cc966419.aspx#mainSection)

**Statistics on Computed Columns**

SQL Server 2005 supports the creation, update, and use of statistics on computed columns even when a query doesn't contain the computed column by name, but rather contains the computed column expression. SQL Server 2000 can only automatically create, update, and use statistics on computed columns if the column is named explicitly in the query.

You can observe auto creation of computed column statistics for the AdventureWorks database table column Sales.SalesOrderHeader.TotalDue if you execute the following Transact-SQL script in SQL Server 2005:

USE AdventureWorks

GO

-- Remove all statistics for Sales.SalesOrderHeader

DECLARE c CURSOR FOR

SELECT name FROM sys.stats

WHERE object\_id = object\_id('Sales.SalesOrderHeader')

AND auto\_created <> 0 AND user\_created <> 0

DECLARE @name NVARCHAR(255)

OPEN c

FETCH next FROM c INTO @name

WHILE @@FETCH\_STATUS = 0

BEGIN

EXEC ('drop statistics Sales.SalesOrderHeader.' + @name)

FETCH NEXT FROM c INTO @name

END

CLOSE c

DEALLOCATE c

-- Query Sales.SalesOrderHeader with an expression equivalent

-- to the TotalDue computed column,

-- ((isnull(([SubTotal]+[TaxAmt])+[Freight],(0))).

SELECT \*

FROM Sales.SalesOrderHeader h, Sales.SalesOrderDetail d

WHERE h.SalesOrderID = d.SalesOrderID

AND (isnull(([SubTotal]+[TaxAmt])+[Freight],(0))) > 200000.00

ORDER BY TotalDue DESC

-- List the created statistics. Observe that statistics

-- are created for TotalDue even though it is not explicitly

-- referenced in the query.

sp\_helpstats 'Sales.SalesOrderHeader'

SQL Server 2005 does not support statistics on non-persisted computed columns that involve a CLR (common language runtime) expression, such as an invocation of a CLR user-defined function. To create statistics on a CLR computed column, the column must be marked PERSISTED.

[Top Of Page](http://technet.microsoft.com/en-us/library/cc966419.aspx#mainSection)

**Statistics on CLR User-Defined Type Columns**

SQL Server 2005 supports the creation, update, and use of statistics on CLR user-defined type columns if the type supports binary ordering. Statistics are not supported on user-defined types that don't support binary ordering. A type is binary ordered if it has the IsByteOrdered flag set to true in the SqlUserDefinedType attribute that is specified as part of the type definition. If a type supports binary ordering that means that standard binary sort ordering is a semantically correct order for the type.

[Top Of Page](http://technet.microsoft.com/en-us/library/cc966419.aspx#mainSection)

**Statistics and Indexed Views**

Normally, statistics are not required on indexed views. This is because the substitution of the indexed views into the query plan is considered only after all the statistics for the underlying tables and indexes are attached to the query plan. There is one exception: statistics will be used if the view is directly referenced in the FROM clauses using the NOEXPAND hint. Note that an error will be generated and the plan will not be created if the NOEXPAND hint is used on a view that does not also contain an index.

Because of their limited use, the statistics on indexed views are not created using **sp\_createstats** or updated using **sp\_updatestats.** The **auto update** and **auto createstatistics** features work for indexed views. But as noted before, such statistics would be required by the optimizer and subsequently created only if the indexed view is used with the NOEXPAND hint in the query and the auto update/create statistics option is turned ON. You can also manually perform CREATE STATISTICS on the indexed view columns or use UPDATE STATISTICS to update column or index statistics on indexed views.

[Top Of Page](http://technet.microsoft.com/en-us/library/cc966419.aspx#mainSection)

**Best Practices for Managing Statistics**

The goal for using statistics in SQL Server is to allow the optimizer to get good cardinality estimates so it can find good query plans, while at the same time keeping any overhead or delay associated with gathering statistics reasonable. Below we list best practices for managing statistics in SQL Server, with the most important ones first.

**Use auto create and auto update statistics**

For a large majority of SQL Server installations, the most important best practice is to use auto create and auto update statistics database-wide. Auto create and auto update statistics are on by default. If you observe bad plans and suspect that missing or out of date statistics are at fault, verify that auto create and auto update statistics are on.

**If needed, selectively use FULLSCAN statistics**

If you are using auto create and auto update statistics and you are getting a bad query plan because the statistics are not accurate or current, do the following:

* leave auto create and auto update statistics on, and
* only for those statistics that are not accurate or current enough, use  
  CREATE STATISTICS … WITH FULLSCAN, NORECOMPUTE  
  and a batch job that does  
  UPDATE STATISTICS … WITH FULLSCAN, NORECOMPUTE periodically.

The frequency with which you should update statistics depends on your application and may require some experimentation to determine. A good starting point for the frequency of **fullscan** update is that if the table under consideration has a high update rate, run **fullscan** statistics update nightly. If the table has a low update rate, run **fullscan** statistics update weekly.

**Avoid use of local variables in queries**

If you use a local variable in a query predicate instead of a parameter or literal, the optimizer resorts to a reduced-quality estimate, or a guess for selectivity of the predicate. Use parameters or literals in the query instead of local variables, and the optimizer typically will be able to pick a better query plan. For example, consider this query that uses a local variable:

declare @StartOrderDate datetime

set @StartOrderDate = '20040731'

select \* from Sales.SalesOrderHeader h, Sales.SalesOrderDetail d

WHERE h.SalesOrderID = d.SalesOrderId

AND h.OrderDate >= @StartOrderDate

The number of rows from Sales.SalesOrderHeader that the optimizer estimates will qualify vs. the condition h.OrderDate >= @StartOrderDate is 9439.5, which is exactly 30% of the size of the table. You can use the graphical showplan for the query and right-click the plan node for Sales.SalesOrderHeader to display this cardinality estimate. In a pre-release version of SQL Server 2005 used while preparing this paper, the plan chosen uses a merge join (the observations that follow are based on this same SQL Server 2005 version; your results may differ depending on your SQL Server version, available memory, etc.). Now, consider this equivalent query that doesn't use a local variable:

SELECT \* FROM Sales.SalesOrderHeader h, Sales.SalesOrderDetail d

WHERE h.SalesOrderID = d.SalesOrderId

AND h.OrderDate >= '20040731'

The cardinality of the result set for the predicate "h.OrderDate >= '20040731'" is estimated as 40 in the graphical showplan for the query (right-click the **filter** operator), for a selectivity of 0.13%. The plan chosen for this query uses a nested loop join instead of a merge join because of this improved estimate.

Even when local variables are used in a query, an estimate that is better than a guess is used in the case of equality predicates. Selectivity for conditions of the form "*@local\_variable = column\_name*" is estimated using the average value frequency from the histogram for *column\_name*. So, for example, if the column *column\_name* contains all unique values, then a selectivity estimate of 1/(number of unique values in column) will be used, which is accurate.

To force SQL Server to optimize a query every time it is run, and use the values of local variables to estimate cardinality and cost during optimization of the query, add the RECOMPILE query hint to the query. For example, modify the first of the two preceding example queries shown to:

SELECT \* FROM Sales.SalesOrderHeader h, Sales.SalesOrderDetail d

WHERE h.SalesOrderID = d.SalesOrderId

AND h.OrderDate >= @StartOrderDate

OPTION(RECOMPILE)

This eliminates the problem of bad estimates due to local variables, at the cost of recompiling the query every time you run it.

To eliminate the use of local variables, consider (1) rewriting the query to use literals directly in queries instead of assigning them to variables and then using the variables in the queries, (2) using **sp\_executesql** or API server cursors with parameters that replace your use of local variables, or (3) using a stored procedure with parameters that replace your use of local variables.

With respect to (1), if the literals do not change between successive submissions of the query, then typically you don't need to take any additional action. However, if you are using dynamically generated SQL and the literals change with every submitted batch, compilation overhead can be quite high, especially for frequently-run, small queries. To avoid this overhead, further modify your application to use parameterized batches or stored procedures instead of dynamic SQL.

**Consider limiting use of multi-statement TVFs and table variables**

Multi-statement table-valued functions (TVFs) have no statistics. The optimizer must guess the size of their results. Similarly, table variables don't have statistics, and the optimizer must resort to guesses for their cardinality. If a bad plan results because of these guesses, consider using a standard table or temporary table as a temporary holding place for the results of the TVF, or a replacement for the table variable. This will allow the optimizer to use better cardinality estimates.

**Non-foldable expressions and built-in scalar functions may result in guesses**

SQL Server can evaluate some expressions that contain only constants at compile time. This is called *constant folding*. Expressions that can be folded are treated just like literals during selectivity estimation. Expressions that cannot be folded result in guesses during selectivity estimation. For example, consider the following Transact-SQL script. This script populates a UserLog table with 200 rows. Half of the rows have unique UserName values and the other half have the same UserName value, giving a skewed distribution.

IF object\_id('UserLog') IS NOT NULL

DROP TABLE UserLog

GO

CREATE TABLE UserLog (UserName NVARCHAR(255), Action NVARCHAR(1000))

DECLARE @i INT

SET @i = 1

SET nocount ON

WHILE @i <= 100

BEGIN

INSERT UserLog VALUES(suser\_sname(), 'login')

INSERT UserLog VALUES(newid(), 'login')

SET @i = @i + 1

END

The **suser\_sname()** built-in function returns the domain\_name\user\_name for the current Windows user and **newid()** gives a unique value. Now, we'll run two different queries that are equivalent. The first one, shown below, contains the predicate UserName = suser\_sname(). The optimizer has to guess the cardinality of the result, and it guesses 1.98 (you can see this in the EstimateRows attribute in the XML showplan produced due to the SET STATISTICS XML ON directive). This is off by a factor of more than 50 since the real cardinality is 100.

GO

SET STATISTICS XML ON

GO

SELECT \* FROM UserLog WHERE UserName = suser\_sname()

GO

SET STATISTICS XML OFF

GO

The second query is issued as a parameterized query through sp\_executesql.   
The suser\_sname() value is passed in as a parameter instead of appearing in  
the query as an expression.

SET STATISTICS XML ON

GO

DECLARE @UserName NVARCHAR(255)

SET @UserName = suser\_sname()

EXEC sp\_executesql N'SELECT \* FROM UserLog WHERE UserName = @n',

N'@n nvarchar(255)', @UserName

GO

SET STATISTICS XML OFF

GO

This time, the optimizer uses an accurate estimate of 50% for the selectivity of UserName = @n. If you look at the XML showplan produced, you'll see that EstimateRows is 100, which is perfect. In a larger and more complex query on a larger data set, this type of error can result in bad plan selection. If this is a problem for your application, consider using a technique like the one illustrated above. Use **sp\_executesql** or a stored procedure containing the problem query, and pass in the precomputed result of the non-foldable expression as a parameter. This will allow you to work around the problem and get good cardinality estimates.

**Use multi-column statistics when queries have multi-column conditions**

When a query has a multi-column condition, consider using multi-column statistics if you suspect that the optimizer is not producing the best plan for the query. You get multi-column statistics as a by-product of creating a multi-column index, so if there is already a multi-column index that supports the multi-column condition, there is no need to create statistics explicitly. Auto create statistics only creates single-column statistics, never multi-column statistics. So if you need multi-column statistics, create them manually, or create a multi-column index

Consider a query that accesses the AdventureWorks.Person.Contact table, and contains the following condition:

FirstName = 'Catherine' AND LastName = 'Abel'

To make selectivity estimation more accurate for this query, create the following statistics object:

CREATE STATISTICS LastFirst ON Person.Contact(LastName,FirstName)

This statistics object will be useful for queries that contain predicates on LastName and FirstName, as well as LastName alone. In general, the selectivity of a predicate on any prefix of the set of columns in a multi-column statistics object can be estimated using that statistics object.

For a statistics object to fully support a multi-column condition, a prefix of the columns in the multi-column statistics object must contain the columns in the condition. For example, a multi-column statistics object on columns (a,b,c) only partially supports the condition a=1 AND c=1; the histogram will be used to estimate the selectivity for a=1, but the density information for c will not be used since b is missing from the condition. Multi-column statistics on (a,c) or (a,c,b) would support the condition a=1 AND c=1, and density information could be used to improve selectivity estimation.

**Be alert for any situation where SQL Server falls back to guessing selectivity**

SQL Server guesses selectivity in a number of situations. Often, the guesses are reasonable, the data sizes are small enough that this is not a problem, or the guess doesn't lead to a bad query plan. Nevertheless, when SQL Server must guess selectivity of a query predicate, sometimes less-than-optimal query plans result. If a query is not performing as well as you would like, and you suspect that a suboptimal query plan is being selected, look at the query and its resulting plan for signs that selectivity was guessed rather than estimated based on statistics. In many cases, you can modify the query or a small part of your application to avoid the guess. The following list gives some situations that can lead to guesses, and possible workarounds:

* Missing statistics: Check if auto create statistics is enabled or make sure to manually create statistics using CREATE STATISTICS or **sp\_createstats.** Check to see if the database is read-only, preventing auto create statistics from working.
* Use of local variables in the query condition (this was addressed previously).
* Non-constant-foldable expressions in the query condition (e.g., T.x+1 = 0, suser\_sname() = T.UserName). Rewrite the query to eliminate the expression, or evaluate the expression prior to executing the query and pass the result into the query as a parameter (not a local variable). In the case of T.x+1 = 0, rewrite the expression as T.x = -1. This is equivalent, and allows an accurate estimate instead of a guess.
* Complex expressions such as "Price + Tax > 100" or "Price \* (1+TaxRate) > 100". If you see slower-than-desired query performance in such a case, consider creating a computed column with the equivalent expression, and creating statistics or an index on the computed column. Auto create statistics will also create statistics for the computed column if it exists, so you need not create the computed column statistics manually if auto create statistics is enabled.

**Avoid modification of stored procedure parameters in an SP prior to use in a query**

For best query performance, in some situations you'll need to avoid assigning a new value to a parameter of a stored procedure within the procedure body, and then using the parameter value in a query. The stored procedure and all queries in it are initially compiled with the parameter value first passed in as a parameter to the query. This is sometimes called *parameter sniffing.* Consider the following stored procedure, which gets sales on or after a certain date, or if NULL is passed as an argument, the last three months of sales:

CREATE PROCEDURE GetRecentSales (@date datetime) WITH RECOMPILE AS

BEGIN

IF @date IS NULL

SET @date = dateadd("mm",-3,(SELECT MAX(OrderDATE)

FROM Sales.SalesOrderHeader))

SELECT \* FROM Sales.SalesOrderHeader h, Sales.SalesOrderDetail d

WHERE h.SalesOrderID = d.SalesOrderID

AND h.OrderDate > @date

END

This SP, if called with NULL, will have the final SELECT statement optimized for @date = NULL. Since no rows have NULL OrderDate, the cardinality estimate for the result of applying this filter to SalesOrderHeader is very low (1 row). However, at run time, the date is not NULL, but three months before the latest OrderDate. The actual number of SalesOrderHeader rows that qualify is 5,736. The optimizer chooses a nested loop join for the query when NULL is passed to GetRecentSales, whereas the optimal plan contains a merge join. You can see the plan selected, and the expected and actual cardinalities, using this script:

SET STATISTICS PROFILE ON

GO

EXEC GetRecentSales NULL

GO

SET STATISTICS PROFILE OFF

GO

Notice that the WITH RECOMPILE option specified on the GetRecentSales stored procedure above does not eliminate the cardinality estimation error. One way to ensure that the queries in this example are optimized with appropriate parameter values that allow good estimates to be obtained is to modify the stored procedure as follows, breaking it down into parts:

CREATE PROCEDURE GetRecentSales (@date datetime) AS

BEGIN

IF @date IS NULL

SET @date = dateadd("mm",-3,(SELECT MAX(OrderDATE)

FROM Sales.SalesOrderHeader))

EXEC GetRecentSalesHelper @date

END

CREATE PROCEDURE GetRecentSalesHelper (@date datetime) WITH RECOMPILE AS

BEGIN

SELECT \* FROM Sales.SalesOrderHeader h, Sales.SalesOrderDetail d

WHERE h.SalesOrderID = d.SalesOrderID

AND h.OrderDate > @date -- @date is unchanged from compile time,

-- so a good plan is obtained.

END

**Consider more frequent statistics gathering for ascending keys**

Ascending key columns, such as IDENTITY columns or datetime columns representing real-world timestamps, can cause inaccurate statistics in tables with frequent INSERTS because new values all lie outside the histogram. Consider updating statistics on such columns frequently with a batch job if your application seems to be getting inadequate query plans for queries that have a condition on the ascending key column. How often to run the batch job depends on your application. Consider daily or weekly intervals, or more often if needed for your application.

**Use asynchronous statistics update if synchronous update causes undesired delay**

If you have a large database, and an OLTP workload, then with AUTO\_UPDATE\_STATISTICS enabled, some transactions that normally will run in a fraction of a second may very infrequently take several seconds or more because they cause statistics to be updated. If you want to avoid the possibility of this noticeable delay, enable AUTO\_UPDATE\_STATISTICS\_ASYNC. For workloads with long-running queries, getting the best plan is more important than an infrequent delay in compilation. In such cases, use synchronous rather than asynchronous auto update statistics.

[Top Of Page](http://technet.microsoft.com/en-us/library/cc966419.aspx#mainSection)

**Summary**

SQL Server 2005 contains many enhancements to its statistics management capabilities. Most importantly, you can rely on the automatic creation and update of statistics to help ensure good query plans in most cases. When automatic statistics with the default sampling rate are not sufficient, you can control statistics sampling rates, and creation and update time, explicitly. If you are observing what appears to be a suboptimal query plan related to statistics or cost estimation, consider the best practices described in this paper.

[Top Of Page](http://technet.microsoft.com/en-us/library/cc966419.aspx#mainSection)

**References**

[Mar04] Arun Marathe, *Batch Compilation, Recompilation, and Plan Caching Issues in SQL Server 2005*,  
[http://www.microsoft.com/technet/prodtechnol/sql/2005/recomp.mspx](http://technet.microsoft.com/en-us/library/cc966425.aspx), July 2004.

[Top Of Page](http://technet.microsoft.com/en-us/library/cc966419.aspx#mainSection)