



# Decoding Captured “Missing Statistics” In ASE 15.0.3 ESD#1

*This article discusses the new feature of ‘capturing missing statistics’, which has been made available in ASE 15.0.3 ESD#1. This is especially relevant for anyone working on query performance in ASE 15. In this article, we’ll also see some interesting application of SQL functions, which will be useful for all ASE 15 users.*

**By Rob Verschoor**

## Statistics In ASE 15: Important!

The point has been made so many times now that you’re probably bored with it, but let me just mention it one more time: having sufficient and accurate statistics is very important in ASE 15, and in fact, it’s even more important than it was in ASE pre-15. The background to this is that the ASE 15 query optimizer must choose between many more query plan options than were available in pre-15 versions, and it will make those choices based on the available statistics.

Against this background, the natural questions to ask are whether your database has all the statistics it needs and whether the existing statistics are accurate. While ASE 15 provides neither a full nor an easy answer to these questions, some clear improvements have been made compared with 12.5.

First, the built-in function `datachange()` was added in ASE 15, indicating the percentage of rows in a table that have changed, potentially indicating the need for updating the table’s statistics. While certainly interesting, discussing `datachange()` will be left for another occasion.

The second improvement in ASE 15 is the ability to identify columns that play a role in a query but for which no statistics exist, and that’s the topic of this article.

## Introducing ‘Missing Statistics’

In ASE 15, running the command `‘set option show_missing_stats on’` will cause the ASE query optimizer to generate messages indicating columns being referenced in SQL queries, but for which no statistics exist. Those messages come in two types, as follows:

```
NO STATS on column my_tab.col1
NO STATS on density set for
my_tab={col2,col3,col4}
```

The first message mentions a single column, indicating that no histogram exists for this column. The second message mentions multiple columns, indicating no density statistics exist for this combination of columns.

The basic idea is that the DBA can create the statistics reported to be missing. The `‘update statistics’` commands creating the statistics corresponding to the messages above would be:

```
update statistics my_tab (col1)
update statistics my_tab (col2,col3,
col4)
```

Note that by default, these ‘NO STATS’ messages are written to the ASE console (that’s not the same as the ASE errorlog!); enable traceflag 3604 to get the output sent to the client.

Enticing as this sounds, it should be noted that there is no guarantee that creating the statistics reported as ‘miss-



*Working as senior technology evangelist for Sybase, Rob Verschoor’s main focus is on Sybase data management products such as ASE and Replication Server. He can be reached at [robv@sybase.com](mailto:robv@sybase.com)*

"Having sufficient and accurate statistics is very important in ASE 15... (because) the ASE 15 query optimizer must choose between many more query plan options than were available in pre-15 versions."

ing' will actually lead to different or better query plans. The significance of these messages is just that the query optimizer might have considered the statistics, had they existed, and in general it is a good principle to provide as much information to the optimizer as possible.

In practice, a DBA must find a pragmatic approach to act upon these 'NO STATS' message, since there could well be too many of these messages to simply create all statistics. Especially for large tables, that could take an unacceptably long time.

### Let's Make 'show\_missing\_stats' Easier

So far, this article has just been setting the stage for the main topic, so let's turn to that now.

A practical downside of 'set option show\_missing\_stats on' is that the DBA must collect all those NO STATS messages, then sort and aggregate them before then trying to draw some intelligent conclusion about how to follow them up. This is a manual activity leaving many opportunities for improvement.

Indeed, in ASE 15.0.3 ESD#1, things have been made easier with the introduction of a new feature that automatically captures the same information as in the NO STATS messages and stores these into the system table *sysstatistics*. This relieves the DBA of the burden of having to collect the messages himself or herself, since the information now can be extracted from *sysstatistics* with a SQL query. However, you'll need a nifty piece of SQL to do that correctly - as will be shown below.

### Nifty SQL – Let's Bring It On!

First, to enable the automatic capturing of missing statistics information, you need to enable the configuration parameter 'capture missing statistics' by setting it to 1 (default = 0, meaning it's disabled). This parameter is dynamic, so changes take effect immediately.

When enabled, ASE will capture the equivalent of the NO STATS messages (obviously, without having to run 'set option show\_missing\_stats', though this command can still be used independently), and store it into the *sysstatistics* table in the database where the table-with-missing-stats is

located. This also applies to the *master* and *tempdb* databases (though see the notes towards the end with respect to *tempdb*).

The rows in *sysstatistics* holding the captured data are identified by the *formatid* column containing the value 110. However, when selecting those rows, you'll see that the data does not resemble anything like column names or column IDs. This is because the column IDs are packed together as 2-byte quantities in the *colidarray* column, making the data rather difficult to query directly.

Fortunately, there is an easy solution here in the form of a user-defined SQL function, which is an ideal solution for decoding this information. NB: SQL functions are available only since ASE 15.0.2; since this article is about an enhancement in 15.0.3 ESD#1, SQL functions are supported.

Below is the T-SQL code for function *sp\_decode\_colidarray* which takes a *colidarray* column as input, and returns a string of the corresponding column names. There is something special about this function which I'll discuss at the end of this article. But first, here's the T-SQL code:

```
use sybsystemprocs
go
create function sp_decode_colidarray
    @colidarray varbinary(100), @id int
    returns varchar(1500)
as
    declare @s varchar(1500) -- assuming that's enough
    declare @len int, @colid int, @colname longsysname

    set @len = datalength(@colidarray)

    while @len > 0
    begin
        set @colid = convert(tinyint, substring(
            @colidarray, @len, 1))
        set @colname = col_name(@id, @colid)
        set @s = @colname + case @s when NULL
            then NULL else ', ' end + @s

        set @len = @len - 2
    end
    return @s
go
```

A SQL function doesn't do anything by itself, but must be invoked as part of a query. Below is SQL source code of stored procedure *sp\_decode\_missing\_stats*, which retrieves the captured missing statistics information from *sysstatistics*, calling the SQL function to decode the column IDs. Note how *sp\_autofORMAT* is used to format the results in a readable way.

```

create proc sp_decode_missing_stats
    @tablename varchar(100) = '%'
as
select Dbname = db_name(),
       Tabname = object_name(s.id),
       NrRows = row_count(db_id(), s.id),
       ColumnList = dbo.sp_decode_colidarray(
           colidarray, s.id),
       Captured = moddate,
       Occurs = convert(smallint,c0)
into #missing
from sysstatistics s, sysobjects o
where s.id = o.id
      and object_name(s.id) like @tablename
      and formatid = 110
      and datalength(colidarray) > 0

exec sp_autoformat #missing,@orderby="order by 2"
go

```

By specifying a table name as a parameter to this stored procedure, or specifying a pattern with wildcard characters, you can restrict the output to one or more specific tables. By default, information for all tables is retrieved.

When running this stored procedure after having captured some missing statistics data, output could look like this:

```

1> my_db..sp_decode_missing_stats
2> go

```

Dbname	Tabname	NrRows	ColumnList	Captured	Occurs
mydb	my_table	7800	col_a	May 26 2009 8:26AM	14
mydb	your_table	5	col_b, col_c	May 26 2009 8:23AM	1
mydb	other_table	54998666	col_d	May 19 2009 10:03PM	202

Some remarks about this output:

- The 'ColumnList' column will either show a single column, or a comma-separated list of column names, with similar meaning as for the NO STATS messages in the introduction of this article: for a single column, it means there is no histogram; for a column list, there are no density statistics. To create those statistics, just run **'update statistics'** on the column or on the list of columns.
- The last column indicates the number of times a particular column or combination of columns has been noticed for not having statistics. ASE automatically accumulates this information in the column `sysstatistics.c0`. NB: This number is a measure of how often the optimizer

comes across this (combination of) columns for which no statistics exist.

- The 'Captured' column shows the first time statistics were noticed to be missing for the column or combination of columns.
- The number of rows in each table is displayed along with missing statistics information. This rowcount is useful information to help interpret the missing statistics data: for a smaller table it is more feasible to try and create additional statistics than for a very large one. Also, you may see missing statistics reported for empty tables, such as the MDA tables, where creating additional statistics probably won't make sense.

Note that the NO STATS messages generated by **'set option show\_missing\_stats'** do not provide any clues about the size of the table, so this stored procedure is an improvement. By presenting the missing statistics information this way, it becomes easier to do a meaningful interpretation of the data and to define follow-up actions.

### An Exercise For The Reader...

There is in fact a very interesting follow-up option that should be mentioned. When using this stored procedure in practice, I discovered that, occasionally, there wasn't even an index on a column reported to have no statistics. Such cases are really worth investigating, since it could potentially indicate that an important index is missing.

It is indeed possible to extend this stored procedure to automatically determine whether the reported columns are indexed (and if so, whether as a first or subsequent column). I have found it very useful to include this information when reporting on missing statistics. However, this extension would take too much SQL code for the space available here in ISUG Technical Journal, so this is left as an exercise for the reader. **Hint:** Use the built-in function `index_col()` to determine if a column is part of an existing index. (Keep an eye on the blog area at <http://blogs.sybase.com/database>; I'll try posting a solution here somewhere in the next few months.)

### Recommendations

Once you've upgraded to ASE 15.0.3 ESD#1, I recommend enabling the configuration parameter **'capture missing statistics'**, and leave it enabled for a while during regular operations. Then use the stored procedure `sp_decode_missing_stats` in this article to see what has been captured. Make sure you run this procedure in all your databases.

The question that then needs to be answered is: which of those missing statistics should be created? There is no single clear answer here, but I’ll list a few things to check:

- Try to determine what role the reported columns play in the queries being executed. Are the columns used for joining? Or are they for looking up or filtering rows? This understanding will be needed to help decide whether it is a problem that the statistics do not exist.
- For the queries being executed, are the query plans optimal or does it look as if they could be better?
- Especially for missing histograms (i.e. a single column being reported), check whether the column is part of an index. If not, consider creating an index on these column(s) or adding them to existing indexes.
- The more often statistics are reported missing; the more interesting candidates there are to be created.
- For small tables, it may be easiest to simply create the missing statistics since that will be a quick operation anyway. Then, see if it makes a difference for query plans and query performance. If there is no improvement at all, consider deleting those statistics again: without keeping them actively updated, the statistics will remain in the database, but will be getting ever older. And that’s rarely a good idea.
- For large tables, you need to combine the understanding from all points above to judge whether creating the additional statistics is worth trying. If so, again verify whether query plans improve and if not, consider deleting the statistics again.

Please keep in mind that all of the above applies only when you feel query performance could be, or should be, better. If you’re happy with your current ASE performance, you should perhaps consider spending your time on something more urgent.

### Usage Notes

The information captured in *sysstatistics* is not cleaned up automatically, but keeps accumulating as long as ‘*capture missing statistics*’ is enabled. Should it be necessary to remove these rows from *sysstatistics*, then you will need to delete them yourself. But be very careful to only remove rows with *formatid = 110*: if you delete other rows you may be removing actual histograms! For this reason, don’t manually delete these rows, but create a stored procedure for this. Place this procedure in *sybssystemprocs* (and make sure the

configuration parameter ‘*allow updates to system tables*’ is enabled when the procedure is created; don’t forget to disable it again after creating the procedure).

Note that information on missing statistics on #temporary tables cannot be retrieved with this feature: even though the data is captured in *tempdb..sysstatistics*, the #temporary table will typically have been dropped by the time you’re looking at the captured data, along with the corresponding rows in *sysstatistics*. This is a difference with ‘*set option show\_missing\_stats*’, which does print messages for #temporary tables, including their table name.

For ‘normal’ (non-temporary) tables in *tempdb*, missing statistics information is captured and accessible just as for other databases, except that the data will disappear after an ASE restart.

Keep in mind that the optimizer will only notice statistics to be missing at the moment when it generates a query plan. For query plans that are already stored in the procedure cache or statement cache, no missing statistics information will be generated when executing these query plans.

Lastly, note that this feature does not indicate that existing statistics are out of date, which is not uncommon to occur. To identify ‘old’ statistics, select the *moddate* column from *sysstatistics* where *formatid = 100*.

### Now, About That SQL Function...

Apart from the functionality it implements, there’s another interesting aspect about the SQL function *sp\_decode\_coliddarray* shown in this article. If you look at the code example, you’ll notice that its name has an *sp\_* prefix and that it is created in *sybssystemprocs*. Using a similar mechanism as system stored procedures like *sp\_help*, this function can now be invoked in the context of any database while it exists only in one place. That’s very useful indeed.

The special part about this is that it’s not formally supported: according to the ASE documentation, this behaviour of *sp\_* named objects is only described for stored procedures. However, it appears to work equally well for SQL functions.

Having said that, this behaviour might actually be changed in some future ASE release, possibly allowing an *sf\_* prefix to be implemented to achieve the same effect, but specifically for functions. Nothing is confirmed about that, but if and when that happens, SQL functions relying on the *sp\_* prefix might need to be changed to start with *sf\_* instead of *sp\_*. Until that time however, *sp\_* should work fine for functions. ■