Paper 300-2010

SAS® Presents In-Database Base Procedures in Practice

Scott Mebust, SAS Institute Inc.
Robert S. Ray, SAS Institute Inc.

ABSTRACT

As part of the SAS® In-Database Initiative, SAS® has enhanced a series of Base SAS® procedures (including FREQ, RANK, and SUMMARY/MEANS) to directly leverage descriptive statistical functions supported by our DBMS partners. These enhancements allow the procedures to drive work into the DBMS using SQL generation in order to greatly reduce or eliminate data transfer into SAS, while leveraging the scalability of parallel databases. This paper explains how you engage these capabilities in a SAS program, explains how you identify when and what kind of in-database queries are passed to the DBMS, and provides some best practices on effectively using them. The discussion will include how these in-database-enabled procedures work on multiple DBMS platforms such as Teradata, Oracle, and DB2.

WHAT'S AN IN-DATABASE-ENABLED PROCEDURE?

To know what an in-database-enabled procedure is, let's consider what it isn't. That is, let's consider what classically happens when a SAS procedure analyzes data that's stored remotely in a database management system (DBMS). When analyzing data stored in a DBMS table, SAS procedures perform most of their work on the machine that's running SAS, by transferring data from the DBMS table to local memory and storage. To do so, a procedure first describes the desired data to a SAS/ACCESS® engine, having a connection to the DBMS, and then reads the observations returned through that engine from the DBMS. From the description of the desired variables and observations, the SAS/ACCESS engine creates a SQL query, selecting the columns, as well as qualifying and possibly ordering the rows, necessary to extract the requested data from the DBMS. The query is submitted to the DBMS for execution through a client interface. Query results are then extracted through this interface from the DBMS, with data being transferred from the DBMS to SAS over the network connection between the two systems.

The SAS procedure is largely oblivious to the communication between the SAS system and the DBMS. From the procedure's perspective, reading rows from a DBMS table is performed in the same way as reading observations from a Base SAS data set. Depending upon the data set access pattern of the procedure, the engine might store the data in a utility file, as well as providing it to the procedure. Use of a multiple-pass access pattern causes the engine to spool the data to a local utility file. After reading the data, the procedure can populate data structures in memory and store the manipulated data to disk, if necessary. Complex query result sets can be analyzed in a similar fashion by reading from SQL views instead of tables.

For both the procedure and the SAS programmer, limiting the variables and observations to only those required for the analysis is a good idea, but the volume of data transferred can nonetheless be quite large. The speed of analysis for large amounts of data extracted from a DBMS can be limited by network bandwidth. A network transfer rate can be significantly less than the transfer rate for local storage devices.

So, what is in-database operation and how can a SAS procedure be enabled to do it? In-database operation is a term used to describe the movement of some or all of the work that needs to be done into the DBMS environment. That is, instead of moving the data to be analyzed across the network from the DBMS to SAS, the work is moved from SAS to the DBMS. The general idea isn't new, but the techniques, application, and implementation are. A simple example is the translation of the list of variables on a KEEP statement to the list of DBMS table columns on a SELECT list of the SQL query generated by a SAS/ACCESS engine or the translation of a WHERE data set option to a WHERE clause on the end of that SQL query. A more complicated example is the identification of a portion of a SAS SQL query and translation of that portion to a vendor-specific SQL dialect that can be passed through to a DBMS from PROC SQL. These are both examples of sending work to the DBMS. Having the DBMS do more of the work can limit the size of the result set being returned to SAS and reduce the amount of work that needs to be done within SAS. An in-database-enabled procedure is therefore a SAS procedure that's been modified to allow some or all of its work to be done by the DBMS.

HOW DOES AN IN-DATABASE PROCEDURE WORK?

So how does an in-database-enabled procedure move its work into a DBMS? It expresses some or all of the work it needs to perform in a SQL query that contains, possibly, references to standard SQL descriptive statistical functions and SAS-supplied custom functions that run within the DBMS. The procedure submits this generated SQL query to the DBMS for execution and, if necessary, retrieves the query results for storage or additional processing. The procedure creates a temporary SAS SQL view in the WORK library through which it can submit the query for execution and from which it can retrieve results. The generated SQL query is processed as a SQL statement by PROC SQL running within the SAS dynamic language processor. Depending upon the procedure, the query is expressed in either generic SAS dialect SQL or in a DBMS-specific SQL dialect. If the query is expressed in generic

SQL, it undergoes a translation to a DBMS-specific SQL dialect through the SAS system's implicit pass-through facility before being submitted to the DBMS for execution. If the query is expressed in a DBMS-specific dialect, explicit pass-through is used to submit the query.

Some Base SAS procedures generate queries that perform aggregation and summarization with mathematical and descriptive statistical functions, grouping table rows by either raw or formatted variable values. The intent is to compute useful partial results that can be integrated into the normal data flow of the procedure and greatly reduce the amount of data returned to SAS. The SAS_PUT user-defined function, which is equivalent to the PUT function in SAS and allows the use of SAS style formats within the DBMS, is very useful in reducing the data volume with this type of aggregating query when grouping by formatted values. Other base procedures that do not perform aggregation and summarization generate queries containing ordered analytical functions and enable the DBMS to perform all of their work, returning the results to SAS or leaving them in the DBMS by inserting them directly into an output table.

WHY WOULD I WANT TO USE IN-DATABASE PROCESSING?

What are the advantages to in-database processing? It can be more efficient because the data isn't copied; this reduces input and output costs. The data isn't transferred across a network; this frees the bandwidth for other uses. Analyzing data within the DBMS can speed the analysis because the data can be read at the transfer rate of the DBMS's storage devices, rather than at the network transfer rate. It can make better use of computing resources by transferring the workload to DBMS hardware, which might be significantly more capable than the machine running SAS. This is especially true when, for instance, the machine running SAS is a desktop personal computer and the DBMS is running on a parallel configuration of processors and storage devices. Such DBMS systems can be highly optimized, tuned, and scalable. Analysis performed in the DBMS can also exceed limitations that might be encountered when performing the analysis within SAS on a smaller system.

WHAT DO I NEED TO USE IT?

You'll obviously need a version of SAS, in which procedures have been enabled for in-database processing, running on a supported platform. You'll also need to be working with a supported DBMS, DBMS version, and platform. Finally, both SAS and the DBMS will need to be configured and prepared for in-database operation. (Contact SAS Sales and Support to see which SAS procedures are currently supported for your DBMS.)

In SAS, a system option (as well as LIBNAME statement option) named SQLGENERATION controls the activation of in-database processing for procedures. In the second phase of SAS 9.2, this option can be set to DBMS to enable in-database processing or set to NONE to disable it. Explicitly setting the option on the LIBNAME takes precedence over any system SQLGENERATION option value. If the option is not explicitly set on a LIBNAME statement, in-database processing is controlled through the system option. In the third phase of SAS 9.2 and later versions, the system SQLGENERATION option provides more fine-grained control and can be used to set the behavior of individual procedures and specific SAS/ACCESS engines.

To prepare the database management system for in-database operation, install support for SAS formats if it is available. SAS format support allows conversion within the DBMS of both numeric and character data to character values using formats embedded by SAS. SAS format support enhances both implicit SQL pass-through from PROC SQL and operation of in-database procedures. After installation, make the formats available for use within the DBMS. SAS formats are made available on a per-database basis in a process called publishing. Both intrinsic formats (those that are provided with SAS) and formats defined with PROC FORMAT can be published. Specific instructions for format publishing vary slightly from one DBMS to another. Detailed instructions for format publishing can be found in the in-database-related documentation for each SAS/ACCESS engine.

One more thing you'll need when using in-database processing is a different mindset for your data processing and analysis. If your current SAS jobs first transfer large amounts of data from a remote DBMS into SAS and then run procedures against the local data, then you'll want to consider opportunities for operating against the data without extracting it from the DBMS first. The idea behind in-database processing is to leave the data in the DBMS. Do not move it close to the SAS system for processing. Instead, move the SAS processing closer to the data by working on the data in the DBMS environment. Copying data out of the DBMS eliminates the possibility of in-database operation and any benefit it might offer. So, you'll want to ask yourself, for each step or procedure within a job, if you have to copy the data or if you can leave it in place.

WHICH PROCEDURES ARE ENABLED?

Within Base SAS, specific descriptive statistic, reporting, and utility procedures have been enabled to perform in-database processing. In the second phase of SAS 9.2, the procedures enabled for in-database operation with Teradata are FREQ, MEANS, SUMMARY, and RANK. Also in the second phase of SAS 9.2, a number of procedures from SAS/STAT® (including PROC VARCLUS, SCORE, REG, and PRINCOMP) are enabled for in-database operation with Teradata. (For more information on in-database processing with advanced statistical procedures, see the SAS/STAT documentation.) In the third phase of SAS 9.2, both DB2 and Oracle database management systems are now supported by the in-database-enabled Base SAS procedures. In addition, the TABULATE, REPORT, and SORT procedures are enabled for in-database operation. In future releases of SAS, the capabilities of the currently

enabled procedures, the list of enabled procedures, and the list of supported database management systems will expand.

HOW DO I MAKE IT WORK?

If you've got everything you need and your data resides within the DBMS, then engaging in-database processing within a procedure is simple. Probably the easiest way to get it working is to set the SQLGENERATION system option to DBMS. This setting tells enabled procedures to generate SQL queries, if possible, for all input data sets residing in libraries using supported SAS/ACCESS engines. Supported engines are those used specifically for the TERADATA, DB2, and ORACLE DBMSs. Setting the system option enables SQL query generation for input data sets read from all libraries that are not explicitly otherwise configured. To undo this system option setting and disable in-database processing, set SQLGENERATION to NONE.

Another way to engage in-database processing for procedures is to set SQLGENERATION as an option on the LIBNAME statement. Setting the option this way takes precedence over any system SQLGENERATION option setting. That is, setting the system option does not override the LIBNAME option setting. Like the system option setting, the LIBNAME option setting affects all enabled procedures.

After you've mastered control of in-database processing using these simple option settings, you may want to investigate other settings for the system SQLGENERATION option available in the third phase of SAS 9.2 and later; these can provide greater control over which procedures perform in-database processing or for which engines the SQL is generated.

HOW DO I KNOW IT'S WORKING?

There are a few different ways to find out whether in-database processing is happening. You could examine the DBMS logs or use a tool to examine what queries are being executed within the DBMS. Also, there are good reasons to look at execution from the server side. Focusing on the SAS client side however, the first and simplest way is to set the system MSGLEVEL option to I. This setting provides additional information regarding the behavior of SAS procedures. If in-database operation is occurring, the setting causes a note regarding SQL generation to be printed to the SAS log. In-database operation can, however, be prevented by the use of various LIBNAME and procedure options. If in-database operation is not occurring, then this option allows one or more notes (explaining why SQL wasn't generated) to be printed to the SAS log. However, this option produces the least verbose logging and doesn't tell you what's happening when in-database operation is occurring. That is, it does not tell you what SQL is being generated.

For procedures FREQ, MEANS, SUMMARY, TABULATE, and REPORT (which all generate generic SQL), the currently undocumented SQL_IP_TRACE system option can reveal the SQL query that they generate. This option is intended to reveal operational details of the SQL implicit pass-through facility. When set to SOURCE, the SQL_IP_TRACE option causes the generic query generated by an in-database-enabled procedure to be printed to the SAS log.

For all Base SAS in-database-enabled procedures, the SASTRACE option reveals the SQL query passed to a DBMS through a SAS/ACCESS engine by an enabled procedure when in-database operation is occurring. Setting this option to ',,,d' and the companion option SASTRACELOC to SASLOG causes the SQL statements passed to the DBMS to be printed to the SAS log. Use of the NOSTSUFFIX option in this context is also recommended, to make the output easier to read. The logging of SASTRACE information can be stopped by setting the SASTRACE option to OFF. Turning on the SASTRACE tracing for an entire session can produce unwanted information that crowds the SAS log. To focus on in-database operation, you can bracket a procedure invocation with these SASTRACE options, to turn the tracing on before the procedure runs and turn it off after the procedure is finished running.

```
OPTIONS sastrace=',,,d' sastraceloc=saslog nostsuffix;
/* Invoke procedure here! */
OPTIONS sastrace=off;
```

WHAT IS IT DOING?

To understand what the in-database procedures are doing when they generate SQL, you can review some examples of the procedures being used and the available logging information produced when they run. Setting the MSGLEVEL and SASTRACE options allows you to know when a procedure is performing its work within the DBMS and what kind of work it is doing. The work being performed within the DBMS is expressed in a SQL query, so some knowledge of the SQL language and its processing is helpful.

For the purposes of demonstration, we'll copy a commonly available data set, SASHELP.CARS, to a Teradata DBMS table, define and publish some SAS formats, run a few procedures using the table as input, and examine the SQL they generate to do work within the DBMS.

The CARS data set, which is found in the SASHELP library, contains information on automobiles manufactured in the model year 2004. To aid in the demonstration, we'll define some formats to categorize values of some of the continuous random variables and make these formats available for use within the DBMS. For comparison, we'll look

at how procedures work without in-database computation and how they work when they are performing in-database computations. When running outside of the DBMS, the SAS/ACCESS engines (through which the procedures communicate with the database systems) generate SQL queries. We'll compare those queries to the queries generated by the procedures when they are running within the DBMS.

Before proceeding, though, we'll take a minute to discuss the general strategies of the procedures and what work they are currently capable of doing within the DBMS.

DO ALL OF THE PROCEDURES OPERATE THE SAME WAY?

The current in-database-enabled Base SAS procedures fall into two sets. One set consists of those procedures that perform aggregation and summarization; the other consists of procedures that perform ranking and sorting. Procedures FREQ, SUMMARY, MEANS, TABULATE, and REPORT are part of the first set; procedures RANK and SORT are part of the second. Because the work they do is similar, producing one or more statistics for each group of observations, the aggregating procedures all share a common SQL code generator. Likewise, the RANK and SORT procedures share a different SQL code generator.

The procedures that perform summarization enlist the DBMS to perform some or all of the summarization work. When possible, all summarization work is completed within the DBMS. If, however, only some of the summarization is performed in the DBMS, then the partially summarized data is returned to SAS for further summarization. These procedures then continue processing the data as they normally would in their regular processing; the data is injected into the data flow of the procedure at an appropriate point. A SQL query produced by one of these procedures looks very similar to the query produced by another of these procedures. The query is characterized by a SELECT statement containing one or more summary (or aggregate) functions and, possibly, GROUP BY and ORDER BY clauses. For these procedures, the SQL is generated in the SAS dialect understood by PROC SQL and is customized for a particular DBMS by the implicit pass-through facility before it is submitted for execution by the SAS/ACCESS engine.

During aggregation for summarization, table rows are usually grouped by the formatted values of TABLE or CLASS variables, if SAS embedded formats are available within the DBMS. Before a procedure uses a format within its generated SQL query, availability of the format is determined by submitting a small query containing the format to the DBMS for preparation and possible execution. If grouping by formatted value is required, but a format is not available for the generated query, it is applied later (within SAS) to the partially summarized data returned from the query and is used to complete summarization. Because of the costs of formatting within the DBMS, applications of formats that will likely result in little aggregation, such as BEST12, can be deferred and performed within the procedure instead of within the DBMS.

The RANK and SORT procedures express the entirety of their processing in SQL, using recent additions to the ANSI SQL standards that are not understood by PROC SQL. Because all of the required work is expressed in SQL, the results do not need to be injected for further processing into some mid-point of the procedure's data processing. Rather, the results can be returned to the end point of the procedure's processing, where they are written to the output data set. Or, the results need not be returned to SAS at all; they can be inserted directly into the output table within the DBMS.

The form and content of the SQL generated by a procedure corresponds to the statements, variables, and options specified when invoking the procedure. Examining a few examples should help to demonstrate this correspondence.

CAN YOU SHOW ME A FEW EXAMPLES?

Executing the following code creates the environment for the in-database procedure examples. It assumes that Base SAS, SAS/ACCESS Interface to Teradata, and an account on a Teradata DBMS are available. In this code, the system SQLGENERATION option is used to enable in-database processing by the procedures.

```
/*------------------------------------------------------------------*/
/* Establish libref / connection to DBMS                            */
/*------------------------------------------------------------------*/
%LET DBMS_ENGINE= teradata;
%LET DBMS_CONNSTR= server=kaching2 user=sas password=sas database=sas;
LIBNAME dbms &DBMS_ENGINE &DBMS_CONNSTR;

/*------------------------------------------------------------------*/
/* Load example DATA into the DBMS                                  */
/*------------------------------------------------------------------*/
DATA dbms.cars;
 SET sashelp.cars;
RUN;

/*------------------------------------------------------------------*/
/* Define some useful formats                                       */
/*------------------------------------------------------------------*/
```

```
PROC FORMAT;

 VALUE CarHwy (default=12)
   low-<25  = 'Wasteful'
   25-<29   = 'Reasonable'
   29-high  = 'Economical';

 VALUE CarCity (default=12)
   low-<18  = 'Wasteful'
   18-<21   = 'Reasonable'
   21-high  = 'Economical';

 VALUE CarPwr (default=10)
   low-<182  = 'Weak'
   182-<232  = 'Standard'
   232-high  = 'Powerful';

 VALUE CarEngSz (default=8)
   low-<2.5 = 'Small'
   2.5-<3.6 = 'Medium'
   3.6-high = 'Large';

 VALUE CarWgt (default=10)
   low-<3252  = 'Light'
   3252-<3790 = 'Moderate'
   3790-high  = 'Heavy';

RUN;

/*************************************************************************/
/* Turn on in-database processing, log messages, and tracing             */
/*************************************************************************/
OPTIONS sqlgeneration=dbms;
OPTIONS msglevel=i;
OPTIONS sastrace=',,,d' sastraceloc=saslog nostsuffix;
```

For the procedure examples below, full SAS log output is not shown. For brevity and focus, only specific log messages relevant to understanding in-database operation are presented. The first couple of SQL queries presented are shown as they would appear within the SAS log, but subsequent queries are formatted and color-coded for easier reading.

HOW DOES THE FREQ PROCEDURE OPERATE IN-DATABASE?

Before examining in-database processing, we'll set the system SQLGENERATION option to NONE and look at the SQL generated by the SAS/ACCESS engine for standard SAS processing. We'll start with a simple, one-way frequency table for a single nominal variable from the CARS data set:

```
OPTIONS sqlgeneration=none;

PROC FREQ DATA=dbms.cars;
 TABLE Origin;
RUN;
```

The FREQ Procedure

Origin				
Origin	**Frequency**	**Percent**	**Cumulative Frequency**	**Cumulative Percent**
Asia	158	36.92	158	36.92
Europe	123	28.74	281	65.65
USA	147	34.35	428	100.00

Looking in the SAS log at the messages generated by the SAS/ACCESS engine from the SASTRACE option setting, we see that a simple SELECT statement is used to retrieve the value of column ORIGIN for all 428 rows in the table.

The columns listed in the select statement correspond to the variables listed on the TABLE statement in PROC FREQ and are only those necessary to create the frequency table. Aggregating the ORIGIN values and counting the number of occurrences for each value is done within SAS.

```
TERADATA_2: Executed: on connection 2
SELECT "Origin" FROM sas."cars"

TERADATA: trget - rows to fetch: 428
```

Now, for a first example of in-database processing, let's look at the same job running with SQLGENERATION=DBMS. In-database processing is evident from the log note regarding SQL generation. The SASTRACE option setting allows the generated SQL query to be printed to the SAS log, as well.

```
OPTIONS sqlgeneration=dbms;

PROC FREQ DATA=dbms.cars;
 TABLE Origin;
RUN;
NOTE: SQL generation will be used to construct frequency and crosstabulation
tables.

TERADATA_6: Executed: on connection 4
 select COUNT(*) as "ZSQL1", case  when COUNT(*) > COUNT(TXT_1."Origin") then '
' else MIN(TXT_1."Origin") end as "ZSQL2" from "sas"."cars" TXT_1 group by
TXT_1."Origin"

TERADATA: trget - rows to fetch: 3
```

The query is printed in an unformatted manner so, for the sake of discussion, we'll examine a formatted version of it and substitute formatted queries, in place, for the unformatted ones in all subsequent examples.

```
select
  COUNT(*) as "ZSQL1",
  case
    when COUNT(*) > COUNT(TXT_1."Origin")
    then ' '
    else MIN(TXT_1."Origin")
  end as "ZSQL2"
from "sas"."cars" TXT_1
group by TXT_1."Origin"
```

We can see that the generated SQL query is no longer a simple SELECT statement. Rather, the query expresses the desired aggregation of ORIGIN values with a GROUP BY clause, counts the number of occurrences of the ORIGIN values within each group, and determines a representative ORIGIN value for each group. The determination of the representative ORIGIN value is slightly more complicated than simply selecting the ORIGIN column, because it must also account for missing values and the possibility of selecting a representative raw value for a group of formatted values. The generated query contains 2 columns and results in only 3 rows being returned to SAS. The benefit of in-database processing, in this case, is that the aggregation and summarization are performed by the DBMS and the volume of data returned to SAS is reduced from 428 rows of 1 column to 3 rows of 2 columns. Note that if there were many distinct values of the ORIGIN column resulting in many groups and little aggregation, the additional columns required for the summary statistics could easily increase the volume of data returned to SAS and negate one of the benefits of in-database processing.

For a second example of PROC FREQ running within a DBMS, we'll create a two-way frequency table of engine size by the number of engine cylinders. We'll also introduce a formatted variable on the TABLE statement. The format, CARENGSZ, is used to assign the values of a continuous variable, ENGINESIZE, to one of three different categories.

```
PROC FREQ DATA=dbms.cars;
 TABLE EngineSize * Cylinders;
 FORMAT EngineSize CarEngSz.;
RUN;
```

The FREQ Procedure

Table of EngineSize by Cylinders								
EngineSize(EngineSize)	Cylinders(Cylinders)							
Frequency Percent Row Pct Col Pct	3	4	5	6	8	10	12	Total
Small	1 0.23 0.78 100.00	124 29.11 96.88 91.18	3 0.70 2.34 42.86	0 0.00 0.00 0.00	0 0.00 0.00 0.00	0 0.00 0.00 0.00	0 0.00 0.00 0.00	128 30.05
Medium	0 0.00 0.00 0.00	12 2.82 7.27 8.82	4 0.94 2.42 57.14	149 34.98 90.30 78.42	0 0.00 0.00 0.00	0 0.00 0.00 0.00	0 0.00 0.00 0.00	165 38.73
Large	0 0.00 0.00 0.00	0 0.00 0.00 0.00	0 0.00 0.00 0.00	41 9.62 30.83 21.58	87 20.42 65.41 100.00	2 0.47 1.50 100.00	3 0.70 2.26 100.00	133 31.22
Total	1 0.23	136 31.92	7 1.64	190 44.60	87 20.42	2 0.47	3 0.70	426 100.00
Frequency Missing = 2								

For a formatted TABLE variable, the FREQ procedure counts the occurrences of the formatted variable values instead of the raw values. Specifying the CARENGSZ format for the ENGINESIZE variable results in significantly more aggregation and a much smaller cross-tabulation, one containing 21 (3 x 7) cells compared to 301 (43 x 7) cells.

The first thing of note appearing within the SAS log when executing this job is a warning that the use of SAS formats within the DBMS is not possible. This warning does not normally appear within the SAS log but is present here because of the MSGLEVEL system option setting. The SELECT statement preceding this warning is a check for the presence of the CARENGSZ format and the ability to use it with DBMS embedded SAS formatting (the SAS_PUT custom function). The attempt to prepare this SELECT statement for execution within the DBMS fails; this failure indicates that the formatting capability is not available. The error associated with the failure is not something about which you need be alarmed. It is not an error issued by SAS but is simply an artifact of the communication between the SAS/ACCESS engine and the DBMS. The message is present in the log due solely to the SASTRACE option setting.

```
TERADATA_8: Prepared: on connection 3
SELECT SAS_PUT(42, 'CARENGSZ.');

ACCESS ENGINE: ERROR: Teradata prepare: Syntax error: expected something
between '(' and the integer '42'. SQL statement was: SELECT SAS_PUT(42,
'CARENGSZ.');.
WARNING: In-database formatting is not available on the database, due to errors
mentioned above. In-database processing will proceed without it.

NOTE: SQL generation will be used to construct frequency and crosstabulation
tables.

TERADATA_11: Executed: on connection 4

      select
       COUNT(*) as "ZSQL1",
       case
         when COUNT(*) > COUNT(TXT_1."EngineSize")
         then NULL
         else MIN(TXT_1."EngineSize")
       end as "ZSQL2",
       MAX(TXT_1."EngineSize") as "ZSQL3",
       case
```

```
          when COUNT(*) > COUNT(TXT_1."Cylinders")
          then NULL
          else MIN(TXT_1."Cylinders")
        end as "ZSQL4",
        MAX(TXT_1."Cylinders") as "ZSQL5"
      from "sas"."cars" TXT_1
      group by TXT_1."EngineSize", TXT_1."Cylinders"
TERADATA: trget - rows to fetch: 55
```

Regardless of the lack of in-database formatting capability, the procedure still proceeds with in-database processing. A preliminary summarization of the data is performed within the DBMS. Afterward, those results are further summarized within SAS. Normally, without in-database processing, the DBMS would transfer the values of 428 rows by 2 columns (ENGINESIZE and CYLINDERS) to SAS. With in-database processing but without the benefit of in-database formatting, this generated SQL query results in the transfer to SAS of about one-third of the data volume (5 numeric columns by 55 rows. Also of note, the MAX aggregate function is used to compute values for 2 of the 5 columns in the query SELECT list. The GROUP BY clause at the end of the query lists the columns for both TABLE variables, so it is the number of distinct values for the combination of these two variables that dictates the number of aggregate groups and, therefore, the number of result rows that are returned by the query.

Additional aggregation and summarization can be done within the DBMS, however, if the formats defined earlier are made available for use within the DBMS. To do this, the format publishing macros are invoked with WORK.FORMATS as the source format catalog.

```
/*************************************************************************/
/* Publish Formats to the DBMS                                           */
/*************************************************************************/
OPTIONS nonotes;
OPTIONS sastrace=off;

%INDTDPF;

%LET indconn=&DBMS_CONNSTR;

%INDTD_PUBLISH_FORMATS(
  action=drop,
  outdir=%QUOTE(C:\temp),
  fmtcat=work.formats
);

%INDTD_PUBLISH_FORMATS(
  action=replace,
  outdir=%QUOTE(C:\temp),
  fmtcat=work.formats
);

OPTIONS sastrace=',,,d' sastraceloc=saslog nostsuffix;
OPTIONS notes;
```

In this publishing example, any formats with the names of those in the WORK.FORMATS catalog are dropped before new formats are installed. The SASTRACE option is temporarily turned off and some log messages are suppressed for this installation. Diagnostic information for the installation is written to files directed to a directory for temporary files.

Now that the few defined formats have been installed within the DBMS. We can again execute the same FREQ procedure job and examine the differences in operation.

```
PROC FREQ DATA=dbms.cars;
 TABLE EngineSize * Cylinders;
 FORMAT EngineSize CarEngSz.;
RUN;
```

The first difference of note is that both the CARENGSZ and BEST formats were inside the DBMS. The BEST format is used in the generated query to format the values of the CYLINDERS column.

```
TERADATA_13: Prepared: on connection 3
SELECT SAS_PUT(42, 'CARENGSZ.');

TERADATA_14: Executed: on connection 3
SELECT SAS_PUT(42, 'CARENGSZ.');

NOTE: The format CARENGSZ was found and will be used for in-database
processing.

TERADATA_15: Prepared: on connection 3
SELECT SAS_PUT(42, 'BEST.');

TERADATA_16: Executed: on connection 3
SELECT SAS_PUT(42, 'BEST.');

NOTE: The format BEST was found and will be used for in-database processing.

NOTE: SQL generation will be used to construct frequency and crosstabulation
tables.

TERADATA_23: Executed: on connection 4

       select
        COUNT(*) as "ZSQL1",
        case
         when COUNT(*) > COUNT(TXT_1."EngineSize")
         then NULL
         else MIN(TXT_1."EngineSize")
        end as "ZSQL2",
        MAX(TXT_1."EngineSize") as "ZSQL3",
        case
         when COUNT(*) > COUNT(TXT_1."Cylinders")
         then NULL
         else MIN(TXT_1."Cylinders")
        end as "ZSQL4",
        MAX(TXT_1."Cylinders") as "ZSQL5"
       from "sas"."cars" TXT_1
       group by
        cast(SAS_PUT(TXT_1."EngineSize", 'CARENGSZ8.0') as char(8)),
        cast(SAS_PUT(TXT_1."Cylinders", 'BEST12.0') as char(12))

TERADATA: trget - rows to fetch: 11
```

Note that because rows of the input table are aggregated by the formatted value of ENGINESIZE and there are many raw ENGINESIZE values that map to a single formatted value, the representative value of ENGINESIZE for a group is determined as the minimum raw value for the group. Use of the minimum raw value to represent a group of formatted values produces results that are consistent with sorting and BY processing within SAS.

With the additional summarization provided by in-database formatting capabilities, this query results in a data volume that is one-fifth the size of the same FREQ job running within the DBMS, but without in-database formatting. The query result set contains only 11 rows of 5 columns. Of these 11 rows, 10 correspond to the populated, non-zero frequency cells of the cross-tabulation, while the eleventh represents two rows of the input table containing NULL values for the CYLINDERS column and a 1.3 liter ENGINESIZE.

As a final example of in-database processing with PROC FREQ, the following job uses formats CARWGT and CARCITY to categorize a vehicle weight as light, moderate, or heavy, and its city gas mileage as wasteful, reasonable, or economical. This FREQ invocation creates a two-way contingency table of WEIGHT by MPG_CITY and performs a chi-square test of independence between the two variables. The procedure is run within the DBMS using in-database formatting.

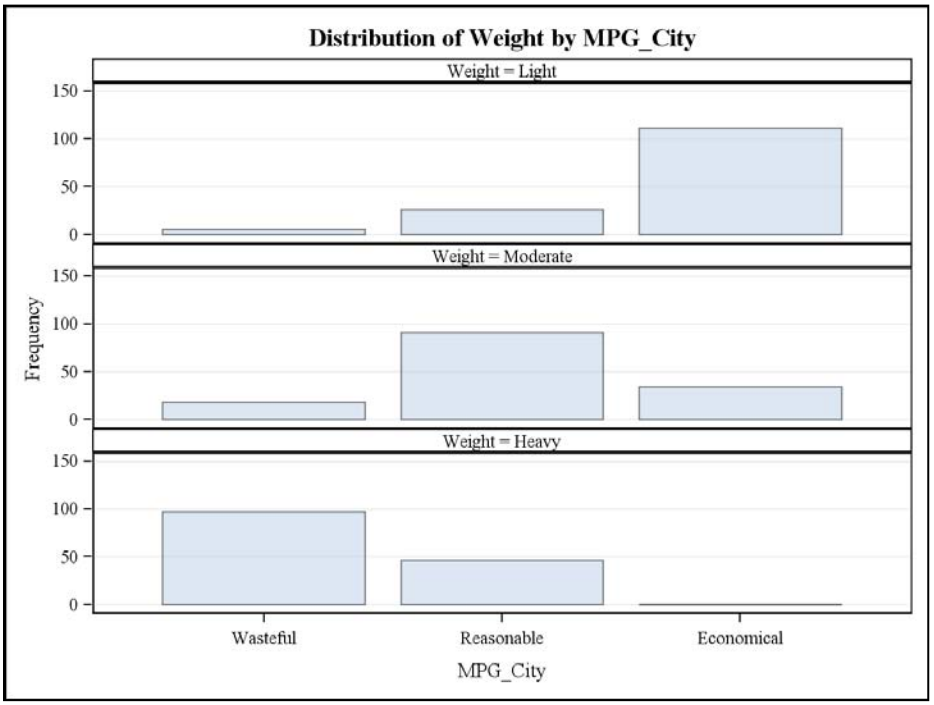
```
ODS GRAPHICS ON;

PROC FREQ DATA=dbms.cars;
 TABLE Weight * MPG_City / CHISQ PLOTS=DEVIATIONPLOT;
 FORMAT MPG_City CarCity.;
 FORMAT Weight CarWgt.;
RUN;
```

```
ODS GRAPHICS OFF;
```

The FREQ Procedure

Table of Weight by MPG_City				
Weight(Weight)	MPG_City(MPG_City)			
Frequency Percent Row Pct Col Pct	Wasteful	Reasonable	Economical	Total
Light	5 1.17 3.52 4.17	26 6.07 18.31 15.95	111 25.93 78.17 76.55	142 33.18
Moderate	18 4.21 12.59 15.00	91 21.26 63.64 55.83	34 7.94 23.78 23.45	143 33.41
Heavy	97 22.66 67.83 80.83	46 10.75 32.17 28.22	0 0.00 0.00 0.00	143 33.41
Total	120 28.04	163 38.08	145 33.88	428 100.00



Statistics for Table of Weight by MPG_City

Statistic	DF	Value	Prob
Chi-Square	4	298.7712	<.0001
Likelihood Ratio Chi-Square	4	323.1764	<.0001
Mantel-Haenszel Chi-Square	1	171.4571	<.0001
Phi Coefficient		0.8355	
Contingency Coefficient		0.6412	
Cramer's V		0.5908	

Sample Size = 428

Examination of the SAS log shows that both CARWGT and CARCITY are found and used for in-database processing. The query result set consists of 8 rows of 5 numeric columns, a significant reduction in data volume from the 428 rows by 2 columns that would result from standard SAS processing.

```
NOTE: The format CARWGT was found and will be used for in-database processing.
NOTE: The format CARCITY was found and will be used for in-database processing.
NOTE: SQL generation will be used to construct frequency and crosstabulation
tables.

TERADATA_35: Executed: on connection 4

      select
       COUNT(*) as "ZSQL1",
       case
        when COUNT(*) > COUNT(TXT_1."Weight")
        then NULL
        else MIN(TXT_1."Weight")
       end as "ZSQL2",
       MAX(TXT_1."Weight") as "ZSQL3",
       case
        when COUNT(*) > COUNT(TXT_1."MPG_City")
        then NULL
        else MIN(TXT_1."MPG_City")
       end as "ZSQL4",
       MAX(TXT_1."MPG_City") as "ZSQL5"
      from "sas"."cars" TXT_1
      group by
       cast(SAS_PUT(TXT_1."Weight", 'CARWGT10.0') as char(10)),
       cast(SAS_PUT(TXT_1."MPG_City", 'CARCITY12.0') as char(12))

TERADATA: trget - rows to fetch: 8
```

The goal of in-database processing with the FREQ procedure is to perform as much of the aggregation and summarization as possible within the DBMS, taking advantage of any parallel processing capabilities of the DBMS, and to reduce the volume of data transmitted from the DBMS to SAS. Toward that goal, it's important to know that the procedure creates a single SQL query containing a GROUP BY clause listing the columns associated with all TABLE variables, for every requested table, and all BY variables. In-database formatting is used, if possible, for all TABLE variables, but not for BY variables. If in-database formatting is not available, the number of distinct combinations of the raw values for all these variables dictates the number rows in the query result set. If in-database formatting is available, the number of distinct combinations of the formatted TABLE variable values and raw BY variable values dictates the number of rows in the result set. Further, it's important to understand that one or more summary statistic columns are returned in the result set for every TABLE variable. Note that if a BY statement has been specified, the input table columns associated with the BY variables are also listed in an ORDER BY clause at the end of the SQL query. The ORDER BY is used to facilitate BY processing during the final aggregation process within SAS. See the Base SAS documentation for the FREQ procedure for more information on the procedure's in-database processing concepts, capabilities, and limitations.

While in-database processing can reduce the volume of data transferred to SAS, it does not necessarily reduce memory requirements of the FREQ procedure. The memory requirement for FREQ is not reduced for in-database processing because the results of the SQL query are used to populate the same in-memory data structures that would have been built in SAS from the original input table.

HOW DO THE SUMMARY AND MEANS PROCEDURES OPERATE IN-DATABASE?

The SUMMARY and MEANS procedures use the same SQL query generator as that used by PROC FREQ. If you understand how in-database processing works with PROC FREQ, you'll feel comfortable with the operation of the SUMMARY and MEANS procedures. In addition, procedures REPORT and TABULATE use the same summarization engine as procedures SUMMARY and MEANS. Therefore, your understanding of their generated SQL and in-database operation will also extend to these two procedures.

For the purposes of demonstration, we'll examine two simple PROC MEANS jobs. The first job specifies only the MIN and MAX statistics for vehicle weight, classifying vehicles by the manufacturer's origin.

```
TITLE "Vehicle Weight, Classified by Origin";

PROC MEANS DATA=dbms.cars MIN MAX;
 VAR Weight;
 CLASS Origin;
RUN;

TITLE;
```

Vehicle Weight, Classified by Origin

The MEANS Procedure

Analysis Variable : Weight Weight			
Origin	**N Obs**	**Minimum**	**Maximum**
Asia	158	1850.00	5590.00
Europe	123	2524.00	5423.00
USA	147	2348.00	7190.00

Without in-database processing, this job would normally result in the transfer of 428 rows, containing only the 2 columns associated with the analysis variable and the class variable, from the DBMS to SAS.

```
SELECT "Weight","Origin" FROM sas."cars"
```

Because there are only three different values of the class variable ORIGIN, in-database processing can greatly reduce the volume of data transferred from the DBMS to SAS. Executing the job with SQLGENERATION=DBMS yields the following:

```
NOTE: SQL generation will be used to perform the initial summarization.

TERADATA_39: Executed: on connection 4

      select
       COUNT(*) as "ZSQL1",
       MIN(TXT_1."Origin") as "ZSQL2",
       COUNT(*) as "ZSQL3",
       COUNT(TXT_1."Weight") as "ZSQL4",
       MIN(TXT_1."Weight") as "ZSQL5",
       MAX(TXT_1."Weight") as "ZSQL6"
      from "sas"."cars" TXT_1
      group by TXT_1."Origin"

TERADATA: trget - rows to fetch: 3
```

Rows are aggregated by the values of the CLASS variable ORIGIN and the statistics requested on the MEANS statement (the minimum and maximum values) are calculated for the analysis variable WEIGHT. In the result set, the column containing the minimum value of WEIGHT is named ZSQL5 and the column containing the maximum value is named ZSQL6. By default, the MEANS procedure also counts the number of non-missing values for the analysis

variable, which appears in the result set as ZSQL4. A representative ORIGIN value for each group appears in the result set as column ZSQL2. Both columns ZSQL1 and ZSQL3 contain a count of the number of rows in each group.

Similar to TABLE variables in the FREQ procedure, all variables listed on the CLASS and BY statements in the MEANS procedure appear in the GROUP BY clause of the generated SQL query. In-database formatting is performed, if possible and practical, for CLASS variables but, again, not for BY variables. The number of calculated columns in the result set for each analysis variable and the aggregate functions used in the generated SQL are directly related to the statistics requested on the MEANS and OUTPUT statements. The default statistics calculated by the MEANS procedure are, of course, N, MEAN, STD, MIN, and MAX.

As another example of the MEANS procedure running within the DBMS, we'll specify two additional statistics, the mean and standard deviation, as well as apply a format to the CLASS variable. In this job, city gas mileage is analyzed across the weight of the vehicle, where the vehicle is classified as a light, moderate, or heavy vehicle.

```
TITLE "City Gas Mileage, Classified by Weight";

PROC MEANS DATA=dbms.cars MIN MAX MEAN STD;
 VAR MPG_City;
 CLASS Weight;
 FORMAT Weight CarWgt.;
RUN;

TITLE;
```

City Gas Mileage, Classified by Weight

The MEANS Procedure

Analysis Variable : MPG_City MPG_City					
Weight	**N Obs**	**Minimum**	**Maximum**	**Mean**	**Std Dev**
Light	142	17.0000000	60.0000000	24.6901408	6.2321326
Moderate	143	12.0000000	23.0000000	19.2727273	1.7287210
Heavy	143	10.0000000	20.0000000	16.2517483	2.0708185

Without in-database processing, the SQL query submitted by the SAS/ACCESS engine to the DBMS would select only the analysis variable and CLASS variable columns from the input table. This would result in 428 observations of 2 columns being transferred to SAS.

```
SELECT "MPG_City","Weight" FROM sas."cars"
```

However, if this MEANS job is processed within the DBMS, the volume of data can be significantly reduced. The table of results transferred to SAS, for the generated SQL query, consists of only 3 observations and 8 columns.

```
NOTE: SQL generation will be used to perform the initial summarization.
NOTE: The format CARWGT was found and will be used for in-database processing.

TERADATA_47: Executed: on connection 4

       select
        COUNT(*) as "ZSQL1",
        MIN(TXT_1."Weight") as "ZSQL2",
        COUNT(*) as "ZSQL3",
        COUNT(TXT_1."MPG_City") as "ZSQL4",
        MIN(TXT_1."MPG_City") as "ZSQL5",
        MAX(TXT_1."MPG_City") as "ZSQL6",
        SUM(TXT_1."MPG_City") as "ZSQL7",
        COALESCE(
         VAR_SAMP(TXT_1."MPG_City") * (COUNT(TXT_1."MPG_City")-1),
         0 ) as "ZSQL8"
       from "sas"."cars" TXT_1
       group by cast(SAS_PUT(TXT_1."Weight", 'CARWGT10.0') as char(10))

TERADATA: trget - rows to fetch: 3
```

In this example, because the formats have been published to the DBMS, the CARWGT format ping succeeds and the format is used within the DBMS to format the values of the WEIGHT column within the GROUP BY clause. The first half of the generated query is very similar to the previous example, with one change being the replacement of the analysis variable WEIGHT with MPG_CITY. The request for the mean statistic causes another column, the SUM of MPG_CITY, to be calculated. Similarly, the request for standard deviation triggers calculation of both the sum and corrected sum of squares (CSS) for MPG_CITY. A CSS function is not commonly available in the SQL understood by most DBMSs, but such a function, if available, will be used directly to calculate this value. If the CSS function is not available and the sample variance exists, CSS(x) is calculated using the sample variance VAR(x):

CSS(x) = VAR(x) * (COUNT(x)-1)

The VAR summary function, known to PROC SQL in SAS, is often translated to VAR_SAMP in the native SQL dialects of DBMSs.

The COALESCE function is used to handle the case when the sample variance is not calculable and is missing. The COALESCE function returns the first non-null value from a list of values or columns. Taken together, the calculation for CSS is:

CSS(x) = COALESCE(VAR(x) * (COUNT(x)-1), 0)

Within the MEANS procedure, the CSS value is used to compute the variance and standard deviation, so the CSS value obtained from the generated SQL is substituted for the CSS value normally calculated by SAS.

Generated SQL for the MEANS and related procedures can express values used to calculate many statistics. The aggregate functions used in generated SQL are limited to COUNT, MAX, MIN, SUM, CSS, and VAR (or VAR_SAMP).

Variance, the second moment descriptive statistic, and the related standard deviation statistic are supported when running within the DBMS. Such higher-moment statistics as skewness and kurtosis are not supported. Many non-weighted statistics and some weighted statistics are supported. Quantile statistics and the mode statistic are not supported when running in-database at this time. For the MEANS and SUMMARY procedures, some statements and options are not supported. See the MEANS and SUMMARY documentation for a complete list of supported and unsupported statistics, statements, and procedure options.

Like the FREQ procedure, the goal of in-database processing with the MEANS, SUMMARY, and related procedures is to perform as much of the aggregation and summarization as possible within the DBMS, taking advantage of any parallel processing capabilities of the DBMS, and to reduce the volume of data transmitted from the DBMS to SAS. These procedures generate a single SQL query containing a GROUP BY clause listing the columns associated with the CLASS and BY variables. In-database formatting is used for CLASS variables, but not for BY variables. If in-database formatting is not available, the number of distinct combinations of the raw values for all of these variables dictates the number rows in the query result set. If in-database formatting is available, the number of distinct combinations of the formatted CLASS variable values and raw BY variable values dictates the number of rows in the result set. If a CLASS statement is specified but no BY statement is present, the number rows in the results set is equivalent to the number of levels in the NWAY combination of class variables. In this case, additional aggregation for other combinations of the class variables is performed using a roll-up process within SAS. It's also important to understand that the volume of data returned in the result set for the in-database operation is proportional to the number of analysis variables.

While in-database processing can reduce the volume of data transferred to SAS, it does not necessarily reduce memory requirements of the SAS procedures. The memory requirements for MEANS and SUMMARY are not reduced for in-database processing because the results of the SQL query are used to populate the same in-memory data structures that would have been built in SAS from the original input table. These structures allow the procedure to complete any roll-ups that are necessary for the TYPES and WAYS statements and also to perform more complex CLASS statement options such as format pre-loading.

HOW DOES THE RANK PROCEDURE OPERATE IN-DATABASE?

The RANK procedure, unlike most of the other enabled procedures, does not perform any form of aggregation and doesn't usually produce an output data set that is smaller than the input data set. Unlike these other procedures, in-database formatting cannot help reduce the volume of data produced by RANK. The size of the output can be smaller than the input, of course, if dropping variables or filtering out observations and it can be equal to the size of the input if replacing analysis variable values with their ranks. It's more likely, though, that new variables are named on the RANKS statement to hold the rank values. When this is the case, the output from PROC RANK will be larger in size than the input. The in-database operation of RANK then is concerned with performing the intensive work of sorting, tied value resolution, ordinal ranking, and the scaling or scoring of the ranks within the DBMS. Nothing can be done to reduce network traffic if the output of the in-database ranking is directed back to SAS. In fact, because of the additional variables in the output, the network transfer will be larger than when not performing in-database processing. Fortunately, in many circumstances, the work of PROC RANK can be done entirely within the DBMS and the results need not be directed back to SAS. If performing in-database processing and directing results to the DBMS, the SQL query generated by RANK can directly populate an output table and eliminate data network traffic

altogether. One extra benefit gained when running within the DBMS is that the RANK procedure is no longer limited by the memory available to SAS, because all of the values within a BY group for a single variable must fit within memory, but can take advantage of the ability of the DBMS to sort large volumes of data.

Ranking the values of a column in a table is not something that is easily done with early revisions of the ANSI SQL standard language, because of the column-oriented nature of the task. Recent ANSI standards for SQL though include extensions for online analytical processing (OLAP) and these include ordered analytical functions that make the task much easier and more efficient. One of the ordered analytical functions is RANK(), but this function produces results that are compatible with only PROC RANK for a specific combination of ranking method and options. Results compatible with other PROC RANK method and option combinations are obtained using the ROW_NUMBER() ordered analytical function, which assigns ordinal ranks, and performing subsequent calculations and manipulations on the value it returns. Ranking within groups, defined by the variables listed on the BY statement, is performed in SQL by specifying PARTITION BY columns within the OVER clause when using ordered analytical functions.

Ranking within the DBMS involves more than just calculating ranks for values in a column. Once determined, the ranks or scores must be somehow combined with the original input table, as either new column values or as replacements for the values of an input column. Composing an output table from the input table and calculated rank values is done using a SQL join. Multiple joins are performed when ranking more than one variable, with one join required for each variable that is ranked. The performance of the DBMS when performing these joins depends upon the abilities of the system's SQL optimizer and factors such as whether the input table is indexed and whether the distribution of the analysis variable values is skewed.

Ranking can be used both in the service of nonparametric statistical tests and simply as a form of scoring or value determination. For examples of in-database ranking, we'll use RANK in the latter context using the data from the CARS data set. As a first example of ranking within the DBMS, take the following RANK job, which determines the most powerful vehicle or vehicles in the data by ranking them on the descending value of HORSEPOWER. In this example, the output is directed to the WORK library.

```
PROC RANK
   DATA=dbms.cars(
     keep =
       Origin
       Make
       Model
       Type
       Horsepower )
   OUT=work.PwrRank
   DESCENDING
   TIES=LOW
   ;
 VAR Horsepower;
 RANKS PowerRank;
RUN;

TITLE "Most Powerful Vehicle(s)";

PROC PRINT DATA=work.PwrRank NOOBS;
 VAR Origin Make Model Type Horsepower ;
 WHERE PowerRank = 1;
RUN;

TITLE;
```

Most Powerful Vehicle(s)

Origin	Make	Model	Type	Horsepower
USA	Dodge	Viper SRT-10 convertible 2dr	Sports	500

With the TIES=LOW option, we see that there is only a single vehicle ranked number one. Examining the SAS log after executing this procedure reveals a relatively large generated SQL query that produces an output result set with 428 rows, the same number of rows contained in the input table. The log also contains a note indicating that ranking was performed within the DBMS.

```
TERADATA_50: Executed: on connection 3

        WITH "subquery0" ( "Horsepower", "Make", "Model", "Origin", "Type" )
        AS ( SELECT "Horsepower", "Make", "Model", "Origin", "Type" FROM "cars" )

        SELECT "table0"."Make", "table0"."Model","table0"."Type",
          "table0"."Origin", "table0"."Horsepower",
          "table1"."rankalias0" AS "PowerRank"
        FROM "subquery0" AS "table0"
        LEFT JOIN (
          SELECT DISTINCT "Horsepower", "tempcol0" AS "rankalias0"
          FROM (
            SELECT "Horsepower",
              MIN( "tempcol1" ) OVER ( PARTITION BY "Horsepower" )
                AS "tempcol0"
            FROM (
              SELECT "Horsepower",
                CAST ( ROW_NUMBER() OVER ( ORDER BY "Horsepower" DESC )
                  AS DOUBLE PRECISION ) AS "tempcol1"
              FROM "subquery0"
              WHERE ( ( "Horsepower" IS NOT NULL ) )
              ) AS "subquery2"
            ) AS "subquery1"
          ) AS "table1"
        ON ( ( "table0"."Horsepower" = "table1"."Horsepower" ) )

TERADATA: trget - rows to fetch: 428
NOTE: SQL generation was used to perform the ranking.
```

This generated query uses a number of features, some of which are available only in SQL processors that conform to recent SQL standards. The first feature of note is the WITH clause, which assigns a nickname to a subquery that otherwise might have to be repeated throughout the query. Restriction of the input to relevant columns and rows can be done in this one place, within the named subquery. In the AS portion of the WITH clause, relevant columns are listed in the SELECT statement and relevant rows are specified with an optional WHERE clause.

The next feature of note is a LEFT JOIN of the restricted input with a derived table containing the calculated ranks of the analysis variable, HORSEPOWER. This join is performed on the values of the analysis variable, a reasonable approach given that there is a one-to-one mapping between values of the analysis variable and the ranks or scores for those values. This approach is not suitable if there is a one-to-many mapping between the values and their ranks. A one-to-many mapping would be required by a tied-value resolution method that, for instance, arbitrarily broke the tie using a random perturbation of the values. Currently, however, the RANK procedure does not implement any tied-value resolution technique that would create a one-to-many mapping.

Another SQL feature is the use of the ROW_NUMBER ordered analytical function, within the innermost subquery of the derived table, to begin the calculation of the ranks. As input to the ROW_NUMBER function, the HORSEPOWER column is first restricted to non-NULL values by a WHERE clause. The non-NULL restriction is necessary because the RANK procedure does not assign any rank to missing values of the analysis variable. The ROW_NUMBER function is then used to assign ordinals to each instance of the descending HORSEPOWER values, in this case because of the DESCENDING procedure option. The assigned ordinals are CAST to a DOUBLE PRECISION type to ensure that tied-value resolution can result in fractional ranks, if necessary. Tied-value resolution is then performed in the surrounding subquery, just one level above. In this example, because TIES=MIN, the MIN() aggregate function is used as a window aggregate function on values of the assigned ordinals, over the subquery results partitioned by HORSEPOWER values. Use of the MIN function in this manner results in the smallest ordinal, among a range of ordinals assigned to a group of HORSEPOWER values being assigned as the rank for the entire group. The outermost subquery that completes the derived table restricts the table to distinct pairs of HORSEPOWER value and corresponding rank. Because this is a one-to-one mapping, the action results in a restriction to distinct values of HORSEPOWER. Distinct values of HORSEPOWER in the derived table are required for the join to produce one output row for every row of the restricted input table.

As another example of in-database ranking, consider a job that determines the least-efficient vehicles in the CARS data set as those vehicles that have the lowest city gas mileage, the lowest highway gas mileage, or both. Further, consider making this determination for each of the three manufacturer origins and directing the results of the RANK procedure back to the DBMS. This example introduces an additional analysis variable and a WHERE option to filter the output of the RANK procedure.

```
PROC RANK
   DATA=dbms.cars(
     keep =
       Origin
       Make
       Model
       Type
       MPG_City
       MPG_Highway )
   OUT=dbms.MpgRank( WHERE = (MpgcRank=1 OR MpghRank=1) )
   TIES=LOW
   ;
 VAR MPG_City MPG_Highway;
 RANKS MpgcRank MpghRank;
 BY Origin ;
RUN;

TITLE "Least Efficient Vehicles By Vehicle Origin";

PROC PRINT DATA=dbms.MpgRank NOOBS;
 VAR Make Model Type MPG_City MPG_Highway;
 BY Origin;
RUN;

TITLE;

PROC SQL;
 DROP TABLE dbms.MpgRank;
QUIT;
```

Least Efficient Vehicles By Vehicle Origin

Origin=Asia

Make	Model	Type	MPG_City	MPG_Highway
Nissan	Pathfinder Armada SE	SUV	13	19
Toyota	Sequoia SR5	SUV	14	17
Toyota	Tundra Access Cab V6 SR5	Truck	14	17
Toyota	Land Cruiser	SUV	13	17
Lexus	LX 470	SUV	13	17

Origin=Europe

Make	Model	Type	MPG_City	MPG_Highway
Land Rover	Discovery SE	SUV	12	16
Land Rover	Range Rover HSE	SUV	12	16
Volkswagen	Phaeton W12 4dr	Sedan	12	19
Mercedes-Benz	G500	SUV	13	14

Origin=USA

Make	Model	Type	MPG_City	MPG_Highway
Hummer	H2	SUV	10	12
Ford	Excursion 6.8 XLT	SUV	10	13

The output of the RANK procedure for this example indicates that multiple vehicles qualify as the least efficient in each region. This is due both to the stated definition of least efficient, as having the worst gas mileage in the city or on the highway, and to some vehicles tying for the worst gas mileage.

One element of in-database processing immediately apparent in the SAS log that, for this example, is different from the previous example, is that no result set is transferred back to SAS. Rather, the generated SQL query is designed to be immediately executed and insert its results into an output table residing in the DBMS. That output table is created, as it would normally be without in-database processing, using a SQL data definition language (DDL) CREATE TABLE command generated by the SAS/ACCESS engine.

```
TERADATA_53: Executed: on connection 3

CREATE MULTISET TABLE sas."MpgRank"
  (
    "Make" CHAR (13),
    "Model" CHAR (40),
    "Type" CHAR (8),
    "Origin" CHAR (6),
    "MPG_City" FLOAT,
    "MPG_Highway" FLOAT,
    "MpgcRank" FLOAT,
    "MpghRank" FLOAT
  )
```

The SQL query generated by the RANK procedure begins with an INSERT INTO, directing the results of the query into the output table. Like the previous example, the query uses features of SQL, such as a WITH clause, ordered analytical functions, window aggregate functions, and windowing defined by the OVER clause that are available only in SQL processors compliant with recent standards.

Similar to the previous example, this query performs a join between the restricted input table and the calculated ranks. Unlike the previous example, this query derives two tables, one for each analysis variable and its associated rank. The first derived table contains MPG_CITY and its rank while the second derived table contains MPG_HIGHWAY and its rank. Consequently, this query must perform two joins, one for each derived table. This query also implements BY processing using the PARTITION BY clause within the window definitions. The PARTITION BY clauses include the column associated with the BY variable ORIGIN. Because ranks are calculated within different ORIGIN groups, the inner subqueries return both the ORIGIN group values as well as the ranks of the gas mileage within the groups. The BY processing complicates not only the partitions within the data window, but also the join conditions.

The join conditions not only have to account for equivalent values of the analysis variable in the restricted input table and the derived table but also values of the BY column that are indistinct. An equality comparison does not suffice for the BY column because a missing value in SAS can compare equally with another missing value. When stored in a DBMS, all SAS missing values, including special numeric missing values, are converted to NULL values. To produce results consistent with SAS processing, the comparison of the columns associated with the BY variables must not only return true when the values are both not NULL and are equal but must also return true in the case that both values are NULL. The SQL comparison expression IS NOT DISTINCT FROM can be used for this purpose, but it is not universally available. Therefore, an equivalent for it is used in this query. The expression ((X = Y) OR (X IS NULL AND Y IS NULL)) is equivalent to the expression X IS NOT DISTINCT FROM Y.

Finally, to accommodate the WHERE data set option on the output data set, the results of the joins are produced in a subquery and then restricted by an equivalent WHERE clause qualifying the outermost SELECT statement. It is the result set of this SELECT statement that is inserted into the output table. For this example, that result set contains 11 rows.

```
TERADATA_54: Executed: on connection 3

INSERT INTO "MpgRank" (
  "Make", "Model", "Type", "Origin", "MPG_City",
  "MPG_Highway", "MpgcRank", "MpghRank" )

WITH "subquery0" (
  "Make", "Model", "MPG_City", "MPG_Highway",
  "Origin", "Type" )
AS (
  SELECT "Make", "Model", "MPG_City", "MPG_Highway", "Origin", "Type"
  FROM "cars"
  )

SELECT "Make", "Model", "Type", "Origin", "MPG_City", "MPG_Highway",
  "MpgcRank", "MpghRank"
FROM (
```

```
SELECT "table0"."Make", "table0"."Model", "table0"."Type",
  "table0"."Origin", "table0"."MPG_City", "table0"."MPG_Highway",
  "table1"."rankalias0" AS "MpgcRank",
  "table2"."rankalias1" AS "MpghRank"
FROM "subquery0" AS "table0"
LEFT JOIN (
  SELECT DISTINCT "Origin", "MPG_City", "tempcol0" AS "rankalias0"
  FROM (
    SELECT "Origin", "MPG_City",
      MIN( "tempcol1" ) OVER ( PARTITION BY "Origin", "MPG_City" )
        AS "tempcol0"
    FROM (
      SELECT "Origin", "MPG_City",
        CAST( ROW_NUMBER() OVER ( PARTITION BY "Origin"
                ORDER BY "MPG_City" )
              AS DOUBLE PRECISION ) AS "tempcol1"
      FROM "subquery0"
      WHERE ( ( "MPG_City" IS NOT NULL ) )
      ) AS "subquery3"
    ) AS "subquery2"
  ) AS "table1"
ON ( ( "table0"."MPG_City" = "table1"."MPG_City" ) AND
     ( ( "table0"."Origin" = "table1"."Origin" ) OR
       ( "table0"."Origin" IS NULL AND "table1"."Origin" IS NULL ) ) )
LEFT JOIN (
  SELECT DISTINCT "Origin", "MPG_Highway", "tempcol2" AS "rankalias1"
  FROM (
    SELECT "Origin", "MPG_Highway",
      MIN( "tempcol3" ) OVER ( PARTITION BY "Origin", "MPG_Highway" )
        AS "tempcol2"
    FROM (
      SELECT "Origin", "MPG_Highway",
        CAST( ROW_NUMBER() OVER ( PARTITION BY "Origin"
                ORDER BY "MPG_Highway" )
              AS DOUBLE PRECISION ) AS "tempcol3"
      FROM "subquery0"
      WHERE ( ( "MPG_Highway" IS NOT NULL ) )
      ) AS "subquery5"
    ) AS "subquery4"
  ) AS "table2"
ON ( ( "table0"."MPG_Highway" = "table2"."MPG_Highway" ) AND
     ( ( "table0"."Origin" = "table2"."Origin" ) OR
       ( "table0"."Origin" IS NULL AND "table2"."Origin" IS NULL ) ) )
) AS "subquery1"
WHERE ( ("MpgcRank" = 1) or ("MpghRank" = 1) )

TERADATA: 11 row(s) inserted/updated/deleted.
NOTE: SQL generation was used to perform the ranking.
```

A SQL query produced by the RANK procedure can appear quite complex, but an understanding of it can be facilitated by first understanding that it contains one derived table joined to the restricted input table for every analysis variable. The whole query becomes easier to comprehend when it is broken into these separate constituent tables because they all follow the same pattern. This one query can be examined, starting from the innermost subquery and proceeding outward. Regardless of how complex one of these SQL queries appears to you, it's how complex it appears to the SQL optimizer that is really important. The complexity of the joins and join conditions, especially the indistinct comparison required for the values of the columns associated with the BY variables, can be difficult for a SQL processor to optimize.

The RANK procedure supports all ranking methods except NORMAL scoring when operating within the DBMS. The join strategy currently used for Oracle precludes dense tied-value resolution, specified by the procedure option TIES=CONDENSE. Due to differences between SAS BY processing and partitioning with PARTITION BY in SQL, the RANK procedure does not support BY processing with formatted variable values within a DBMS. Also, the RANK procedure currently restricts in-database processing to jobs for which no variable is explicitly formatted.

HOW DOES THE SORT PROCEDURE OPERATE IN-DATABASE?

The SORT procedure has, for many years, been able to perform a simple in-database operation by converting a request to sort a data set, with the help of the SAS/ACCESS engine, into a SQL SELECT statement with an ORDER BY clause. The capability of translating a sorting request in this fashion is also used by the greater SAS system to perform an implicit sort when a BY statement is encountered in the context of a DATA Step or a procedure. For the SORT procedure, few deviations from a simple sort are allowed when performing this simple in-database operation. Notably, the NODUPKEY option disallows this form of in-database processing because no equivalent ORDER BY option exists in SQL and the SQL language does not otherwise facilitate this type of operation. With the advent of more recent SQL standards, including OLAP extensions, and support by vendors for these standards, SORT has now been extended with the capability of performing NODUPKEY processing within a DBMS.

To implement NODUPKEY processing, SORT uses some of the same SQL constructs as the RANK procedure uses to perform its work in a DBMS. Specifically, SORT uses the ROW_NUMBER() ordered analytical function to express duplicate key elimination as a ranking and filtering problem. Unlike RANK, the SORT procedure with the NODUPKEY option is capable of reducing the volume of data that is transferred to SAS. If the number of duplicates keys is large or the number of BY groups is small, the number of observations in the result set will be small and the volume of data transferred to SAS will likewise be small. Even so (like RANK), SORT can directly populate an output table on the DBMS with its results.

A simple example of the SORT procedure's new ability to run in-database should suffice to demonstrate the SQL features used in the generated SQL. The following SORT job, using the NODUPKEY option, processes the CARS data set and eliminates observations having duplicate values for the combination of the BY variables DRIVETRAIN and CYLINDERS. For simplicity, the input is restricted to just these two variables, but this need not be the case. In this example, the SORT procedure's output data set is directed to the WORK library, but it could just as easily be directed back to the DBMS. The output of the procedure is then printed for visual examination.

```
PROC SORT NODUPKEY
   DATA=dbms.cars(keep=DriveTrain Cylinders)
   OUT=work.DrvCyl;
 BY DriveTrain Cylinders;
RUN;

PROC PRINT DATA=work.DrvCyl;
RUN;
```

Obs	DriveTrain	Cylinders
1	All	4
2	All	5
3	All	6
4	All	8
5	All	10
6	Front	3
7	Front	4
8	Front	5
9	Front	6
10	Front	8
11	Front	12
12	Rear	.
13	Rear	4
14	Rear	6
15	Rear	8
16	Rear	10
17	Rear	12

The output of the procedure contains 17 observations, one of which appears to show a rear-wheel drive vehicle that does not use a standard combustion engine. The SQL query generated for this SORT job in some ways resembles the queries produced by the RANK procedure. Like those queries, this one uses features that appear in relatively recent SQL standards. These features include the WITH clause, the ROW_NUMBER ordered analytical function, and an OVER clause including both PARTITION BY and ORDER BY clauses. The WITH clause is not necessary because the subquery it names does not appear or have the potential to appear in multiple locations within the query. However, it does make a convenient place to restrict the input table using the SELECT statement and WHERE clause, when necessary. Examining the query from the innermost subquery outward reveals two parts to duplicate key elimination. The first part, expressed in the inner query, partitions and orders the rows of the restricted input table and assigns ordinals to the rows within each partition. That is, each partition is the equivalent of a BY group, the first row within a partition is the first observation within that group, and that first row is assigned the ordinal one. The second row within a partition is assigned the ordinal two, the third row is assigned ordinal three, and so on. The rows identified by the ordinal one are those that are preserved in the final result set. This is similar to SORT eliminating all but the first observation within a BY group. For SAS data sets, which have an inherent physical ordering of observations, the first observation encountered within a BY group during a sort is also the first observation encountered within the input data set if the sort being performed is stable. Sort stability, specified by the EQUALS procedure option and the SORTEQUALS system option, is the default for the SORT procedure in SAS. For input data sources (such as a DBMS) that might deliver observations to a SAS procedure in a non-deterministic order, the selection of which observation to keep and which duplicates to eliminate can appear to be arbitrary. The syntax of the SORT procedure does not currently allow for ordering by additional variables, other than those specified on the BY statement, to force the selection of a particular observation based on the values of those additional variables. However, should the syntax of the SORT procedure change in the future, the ORDER BY clause that follows the PARTITION BY clause allows for such control to be expressed within the SQL query. The subquery just outside of the innermost query restricts the result set, using a WHERE clause, to only those rows that were assigned the ordinal one. The duplicate key elimination problem posed to SORT is expressed, then, as a ranking and filtering problem; the inner query assigns ranks using the ROW_NUMBER function and the outer query filtering the result set based on those ranks.

```
TERADATA_64: Executed: on connection 3

      WITH "subquery0" ( "Cylinders", "DriveTrain" )
      AS ( SELECT "Cylinders", "DriveTrain" FROM "cars" )
      SELECT "table0"."DriveTrain", "table0"."Cylinders"
      FROM (
        SELECT "Cylinders", "DriveTrain"
        FROM (
          SELECT "Cylinders", "DriveTrain",
            ROW_NUMBER() OVER (
              PARTITION BY "DriveTrain", "Cylinders"
              ORDER BY "DriveTrain", "Cylinders"
              ) AS "tempcol0"
          FROM "subquery0"
          ) AS "subquery1"
        WHERE ( "tempcol0" = 1 )
        ) AS "table0"
      ORDER BY "table0"."DriveTrain", "table0"."Cylinders"

TERADATA: trget - rows to fetch: 17
NOTE: SQL generation was used to perform the sorting.
```

The queries produced by the SORT procedure, although created by the same SQL generator used by RANK, are obviously much simpler than those produced by the RANK procedure. There are no joins performed by a SORT query, and there are no complex join conditions that might be hard to optimize. A query generated by SORT is relatively simple, but can still offload a great deal of work to a DBMS.

IS THAT ALL THERE IS TO IT?

That's pretty much all there is to it but to complete this examination of in-database procedure operation, we can submit some code to clean up after ourselves and remove from the DBMS both the formats that were published there and the CARS table that we created.

```
/*************************************************************************/
/* Cleanup                                                               */
/*************************************************************************/
OPTIONS sastrace=off;

OPTIONS nonotes;

%INDTD_PUBLISH_FORMATS(
  action=drop,
  outdir=%QUOTE(C:\temp),
  fmtcat=work.formats
);

OPTIONS notes;

PROC SQL;
 DROP TABLE dbms.cars;
QUIT;

LIBNAME dbms;
```

If you have more questions about what the procedures are doing to perform work within the DBMS or questions about the capabilities and restrictions of the procedures, you'll find more information in the SAS procedure documentation.

SO, WHEN SHOULD I USE IN-DATABASE PROCESSING?

You should consider activating in-database operation of enabled procedures when, to state the obvious, the data you want to analyze resides within a supported DBMS and would normally be transferred, in whole, to SAS through a SAS/ACCESS engine. Furthermore, if the volume of data normally transferred from the DBMS to SAS is large and the network bandwidth is a limiting factor on performance, consider activating in-database operation. You should also consider in-database operation if the DBMS is significantly more powerful than the machine on which SAS is running and especially if the DBMS is configured for, and capable of, highly scalable parallel processing If a query generated by an in-database-enabled procedure can be distributed across the processing units of a parallel DBMS and run on independent data partitions, the time to solution for the SAS procedure can be greatly reduced.

For the summarizing and aggregating procedures, consider in-database operation when you expect effective reduction in data volume. This is likely, for instance, in a batch production job when you know the general qualities of the input data and the expected output. A reduction in data volume within the DBMS means that more work is performed in the DBMS and less data is transmitted across the network. For PROC FREQ, effective reduction occurs when there are few cross-tabulation cells produced by the combination of TABLE variable values and few BY groups produced by the combination of BY variable values, relative to the number of observations being processed. Likewise, for the MEANS, SUMMARY, and TABULATE procedures, effective reduction occurs when there are relatively few CLASS levels and BY groups. For the REPORT procedure, relatively few BY groups will result in effective data reduction. When a BY statement is used in conjunction with either a TABLE or a CLASS statement, the number of distinct value combinations for all of the variables listed on both statements determines the degree of data reduction within the DBMS. For the FREQ procedure, specification of multiple tables (on a single statement or multiple statements) can impede data volume reduction because the procedure creates only one query for the invocation, and the number of groups of values for the combination of variables listed across all tables dictates the degree of data reduction. For those procedures that support the CLASS statement, lack of the CLASS statement implies a single class level and can result in very large reduction in data volume. Likewise, lack of a BY statement implies a single BY group and can also result in a large reduction in data volume.

Use of categorical or nominal variables in the TABLE, CLASS, and BY statements is likely to produce effective data reduction, while use of continuous variables will not. Formatting the TABLE or CLASS variables can be effective in reducing the number of cross-tabulation cells or class levels and facilitating data reduction within the DBMS when those formats are available for use in the DBMS. Formatting of BY variables is not likewise effective in reducing the number of BY groups, because those formats are not applied within the DBMS, so the number of BY groups produced by the raw values determines the degree of data reduction within the DBMS.

For the other procedures, which do not summarize or aggregate data, consider the amount of work that can be performed within the DBMS, whether there is any reduction in data volume, and whether the results remain within the DBMS or are transmitted back to SAS. Both procedures RANK and SORT perform sorting, which is often an expensive operation and can be performed more quickly and more efficiently by the DBMS. While selection of columns and filtering of rows can reduce data volume, the SORT procedure can also reduce data volume through elimination of rows with duplicate key values. If a large data volume reduction due to NODUPKEY processing is expected, in-database operation should be considered. Further, if the results of the procedure are not transmitted

back to SAS, the in-database operation can be very advantageous. Finally, in-database operation should be considered if the DBMS presents fewer limitations for the requested processing. For instance, it might be possible to rank many more values within the DBMS than in SAS, because of the memory limitations inherent in the native processing of the RANK procedure.

WHEN SHOULD I NOT USE IT?

If your data is located on a supported DBMS, you'll probably want to perform standard SAS processing instead of in-database processing, when dealing with small data volumes, if the network bandwidth is not a performance impediment, if the SAS system is reasonably powerful, or when the DBMS is already highly loaded. You'll probably also want to rely on standard SAS processing when the combination of values for formatted TABLE, formatted CLASS, and unformatted BY variables is large relative to the number of observations you're analyzing. When little data volume reduction is gained through aggregation, the volume of data returned from the DBMS to SAS can actually be larger than simply transferring the input data, because of the addition of columns for the summary statistics. For this reason, you probably do not want to use in-database processing for interactive data exploration when the qualities of the input data are not well understood. Finally, you'll want to disable in-database computation, at least temporarily, if you encounter a situation in which the in-database processing appears to take an unexpectedly long time or an error occurs. In this case, determine why processing is not as fast as expected or why the error occurred and whether there is any way to make the in-database processing work better.

HOW DO I MAKE IT WORK BETTER?

You might be able to make in-database processing work better by making adjustments in SAS or in the DBMS. Of course, you'll want to observe a running procedure to see what it is doing and then identify those modifications that can help its operation.

For SAS, first follow common rules of efficient data processing; subset the input data and limit it to the data required for only the procedure and subsequent steps. Using data set options, drop unnecessary variables and keep only those that are necessary. Using either a WHERE statement or data set option, filter the set of observations to process. Do the same for the procedure output, if possible, using the DROP, KEEP, and WHERE data set options.

Second, be aware of what's being done in-database and what might be preventing in-database operation. Use the MSGLEVEL option to see whether the procedure is running within the DBMS and, if not, then what is preventing it from doing so. Look for notes in the SAS log that provide details regarding the in-database operation, such as those that indicate whether formats are available and being used in the DBMS. Publish custom formats to the DBMS if the procedure indicates they are not available. Use the SASTRACE option to determine what SQL query is being passed to the DBMS and understand how the query relates to the procedure and options that you specified.

There are some procedure-level adjustments worth considering. For the FREQ procedure, consider specifying each table in a separate procedure invocation if you've requested statistics for multiple tables. Invoking the procedure multiple times may cause the DBMS to do more work because it will make multiple passes over the input data, but requesting multiple tables within a single invocation can limit aggregation and not allow a suitable reduction in volume of the data transferred back to SAS. For the MEANS, SUMMARY, and TABULATE procedures, try specifying CLASS variables rather than BY variables if a large volume of data is being transferred to SAS as a result of in-database operation. The use of CLASS variables facilitates aggregation if the variables are formatted, because the formats can be applied to the data on the DBMS, while BY variable formatted is performed after the initially aggregated data is returned to SAS. However, if a large amount of memory is required by one of these Base SAS procedures and features of CLASS processing are not required, then the use of BY processing instead of CLASS processing can be beneficial. With BY processing, no internal CLASS structures are built by these procedures and the memory required is reduced to a constant value independent of the data.

For the DBMS, you should understand how well the SQL generated by the SAS procedure is executing. Copy the generated query from the SAS log, and analyze it within the DBMS environment. Use the EXPLAIN facility and DBMS vendor tools to examine the SQL processor's execution plan, determine how the query is satisfied, and observe it during execution. Understand the effects that table partitioning, indexing, column statistics, and data skew have on the plan generated by the SQL optimizer. Consult with your database administrator (DBA) regarding system configuration, as well as table partitioning, indexing, and statistics generation that might improve performance.

WOULD YOU REPEAT THAT?

An in-database-enabled procedure is one that can perform some or all of its work within a DBMS by generating a SQL query, wrapping it in a PROC SQL view, and submitting it to a DBMS for execution. The results of the query either are the answer to the problem posed to the procedure or are partial results that the procedure can use to obtain the final result. You should take advantage of in-database-enabled procedures because they can provide performance gains by offloading the work normally done by SAS to the DBMS, possibly a more powerful and scalable system. Further, using in-database-enabled procedures can provide a speed boost by transferring much less data at network speed between the DBMS and SAS.

In-database-enabled procedures are available in the second phase of SAS 9.2 and later releases for some database management systems. Support for in-database use of SAS formats can be installed, and formats can be published to the DBMS using SAS macros that are specific to each SAS/ACCESS engine and described in the SAS/ACCESS documentation. In the third phase of SAS 9.2, the FREQ, MEANS, SUMMARY, REPORT, TABULATE, RANK, and SORT procedures are in-database-enabled. You can activate in-database processing for procedures using the LIBNAME and system SQLGENERATION. You can also know whether or not they are performing their work in-database and, if not, learn why not by using the MSGLEVEL option an examining the log for in-database related messages.

You can determine exactly what work they are performing within the DBMS by examining the SQL queries they generate by using the SASTRACE system option. The FREQ, MEANS, SUMMARY, REPORT, and TABULATE procedures perform aggregation and summarization within the DBMS. The work of these procedures can be greatly aided through the use of in-database SAS formats. The aggregation and summarization they perform within the DBMS can greatly reduce the amount of data transferred back to SAS. These procedures all share a common SQL code generator. The RANK and SORT procedures rely on the DBMS to do the heavy lifting of sorting, ranking, and scoring. Their result sets might not be smaller than the input, but they can, when their output is directed to the DBMS, avoid any data transfer back to SAS. There are certain times you should definitely take advantage of in-database processing, such as when the data being analyzed can be summarized down to a small fraction of its input size, and times when you might want to avoid it. There are also ways to help in-database processing work smarter, better, and faster when it isn't performing as expected. The key to determining what to do is to understand what work is being performed within the DBMS and why it isn't working as expected. Understanding the performance impediments of the SQL queries submitted to the DBMS can be aided by such things as an EXPLAIN facility and system monitoring tools. Whew.

So where is all of this in-database stuff headed? We at SAS continue to work with DBMS vendors to improve the current in-database capabilities, as well as develop new methods of pushing work into the DBMS. Future work will certainly include bug fixes to the existing line of procedures, and additional procedures are being evaluated for in-database work possibilities. Even some DATA step functionality is being considered for implementation. For the DATA Step, implementing more than some functionality looks like a giant step, so we'll just take that one step at a time.

RECOMMENDED READING

- SAS Institute. 2007. SAS Institute technical paper. "SAS In-Database Processing: A Roadmap for Deeper Technical Integration with Database Management Systems." Available at: http://support.sas.com/resources/papers/InDatabase07.pdf
- SAS Institute. 2008. SAS Institute technical paper. "SAS In-Database Processing with Teradata: An Overview of Foundation Technology." Available at: http://support.sas.com/resources/papers/teradata08.pdf
- Jeff Bailey and Pat Bostic. 2009. "Publish SAS Formats in Your Teradata Server; Impress Your Friends." *Proceedings of the SAS Global Forum 2009 Conference*. Paper 338-2009. Cary, NC: SAS Institute Inc Available at: http://support.sas.com/resources/papers/sgf09/338-2009.pdf
- SAS Institute. 2010. "Deploying and Using SAS Formats in Teradata." *SAS/ACCESS 9.2 for Relational Databases: Reference, Second Edition*. Available at: http://support.sas.com/documentation/cdl/en/acreldb/63023/HTML/default/a003276900.htm
- SAS Institute. 2010. "Base Procedures That Are Enhanced for In-Database Processing." *Base SAS 9.2 Procedures Guide*. Available at: http://support.sas.com/documentation/cdl/en/proc/61895/HTML/default/a003352478.htm

ACKNOWLEDGMENTS

We would like to thank all of the contributors to the in-database initiative at SAS, including developers, testers, documentation specialists, and management. We would also like to thank those people who contributed to this paper and who spent time reviewing its content. Special thanks are offered to David Shamlin, Mike Whitcher, Gordon Keener, Lewis Church, Dale Lloyd, Doug Christie, Kim Sherrill, Laura Gold, Michelle Mebust, Nancy Moser, Amy Wolfe, and Mike Boyd, all from SAS, as well as Michael Watzke, Greg Otto, Brian Mitchell, and others from Teradata.

CONTACT INFORMATION

Your comments and questions are valued and encouraged. Contact the authors at:

Scott Mebust
SAS Institute
SAS Campus Drive, R1437
Cary, NC 27513
(919) 531-2356
Scott.Mebust@sas.com

Robert Ray
SAS Institute
SAS Campus Drive, R1437
Cary, NC 27513
(919) 531-6605
Robert.Ray@sas.com