Paper 036-2013

Big Data, Fast Processing Speeds

Kevin McGowan SAS® Solutions on Demand, Cary NC

ABSTRACT

As data sets continue to grow, it is important for programs to be written very efficiently to make sure no time is wasted processing data. This paper covers various techniques to speed up data processing time for very large data sets or databases, including PROC SQL, data step, indexes and SAS® macros. Some of these procedures may result in just a slight speed increase, but when you process 500 million records per day, even a 10% increase is very good. The paper includes actual time comparisons to demonstrate the speed increases using the new techniques.

INTRODUCTION

More organizations are running into problems with processing big data every day. The bigger the data, the longer the processing time in most cases. Many projects have tight time constraints that must be met because of contractual agreements. When the data size increases, it can mean that the processing time will be longer than the allotted time to process the data. Since the amount of data cannot be reduced (except in rare cases), the best solution is to seek out methods to reduce the run time of programs by making them more efficient. This is also a cheaper method than simply spending a lot of money to buy bigger/faster hardware, which may or may not speed up the processing time.

Important Note: In this paper, whenever code is presented that is efficient it will be shown in green. Code that should **not** be used is shown in red.

WHAT IS BIG DATA?

There are many different definitions of "big data." And more definitions are being created every day. If you ask 10 people, you will probably get 10 different definitions. At SAS® Solutions on Demand (SSO) we have many projects that would be considered big data projects. Some of these projects have jobs that run anywhere from 16 to 40 hours because of the large amount of data and complex calculations that are performed on each record or data point.

These projects also have very large and fast servers. One example of a typical SAS server that is used by SSO has these specifications:

- 24 CPU cores
- 256 Gb of RAM
- 5+ Tb of disk space
- Very fast RAID disk drive arrays with advanced caches
- Linux or AIX operating system
- Very high speed internal network connections (up to 10 Gb per second)
- Encrypted data transfers between servers
- Production, Test, and Development servers that are identical
- Grid computing system (for one large project)

The projects also use a three-tiered system where the main server is supported by an Oracle® database server and a front-end terminal server for the end users. The support servers are typically sized about the same size as the SAS server. In most cases the data is split – some data is stored in SAS data sets while the data used most by the end users is stored in Oracle tables. This setup was chosen because Oracle tables allow faster access to data in real time. Even with this large amount of computing "horsepower," it still takes a long time to run a single job because of the large amount of data the projects use. Most of these projects process over 200 million records during a single run. The data is complex, and requires a large number of calculations to produce the desired results.

BEST WAYS TO MEASURE SPEED IMPROVEMENTS

SAS has several system options that are very helpful to determine the level of increase in performance. In this this paper, we will focus on the actual clock time (not CPU time) the program takes to run. In an environment with multiple CPUs, the CPU time statistic can be confusing – it's even possible that CPU time can be greater than the clock time. The most important SAS options for measuring performance are listed below with a short description:

Stimer/Fullstimer - These options control the amount of data produced for CPU usage. Fullstimer is the preferred option for debugging and trying to increase program speed.

Memrpt - This option shows the amount of memory usage for each step. While memory usage is not directly related to program speed in all cases, this data can be helpful when used along with the CPU time data.

Msglvl=I - this option outputs information about the index usage during the execution of the program.

Options Obs=N - This option can be very useful to test programs on a small subset of data. Care must be taken to make sure that this option is turned off for production

DATABASE / DATASET ACCESS SPEED IMPROVEMENTS USING SQL

Since many SAS programmers access data that is stored in a relational database as well as SAS data sets, this is a key area that can be changed to speed up program speed. In an ideal world, the SAS programmers would be able to help design the database layout. But, that is not always possible. In this paper, we will assume that the database design is complete, and the SAS programmer accesses the data with no ability to change the database structure. There are three main ways SAS developers access data in a relational database:

- PROC sql
- LIBNAME access to a database
- Converting database data to text files (this should be a last resort when no other method works, such as when using custom database software)

Most of the methods described here will work for all three methods for database access. One of the primary reasons to speed up database access is that it is typically one of the easiest ways to speed up a program. Database access normally uses a lot of input/output (I/O) to disk, which is slower than reading data from memory. Advanced disk drive systems can cache data in memory for faster access (compared to disk) but it’s best to assume that the system you are using does not have data cached in memory.

The simplest way to speed access to either databases or SAS data sets is to make sure you are using indexes as much as possible. Indexes are very familiar to database programmers but many SAS programmers, especially beginners, are not as familiar with the use of indexes. Using data without indexes is similar to trying to find information in a book without an index or table of contents. Even after a project has started, it’s always possible to go back and add indexes to the data to speed up access. Oracle has tools that can help a programmer determine which indexes should be added to speed up database access – the system database admins (DBAs) can help with the use of those tools.

There are many, many methods to speed up data access. This paper will list the methods the author has used over the years that have worked well. A simple Google search on the topic of SQL efficiency will find other methods that are not covered in this paper.

DBAs can be very helpful in making database queries run faster. If there is a query or set of queries that is running long, a good first step is to get the DBAs to take a look at the query while it is running to see exactly how the query is being processed by the database. In some cases, the query optimizer will not optimize the query because of the way the SQL code is written. The DBAs can make suggestions about how to rewrite the query to make it run better.

The first method is to **drop indexes and constraints when adding data to a database table**. After the data is loaded, the indexes and constraints are restored. This speeds up the process of data loading, because it's faster to restore the indexes than to update them every time a record is loaded. This is very important if you are importing millions of records during a data load. There is one problem to watch out for with this method – you have to make sure the data being loaded is very clean. If the data is not clean, it could cause problems later when the indexes and constraints are put back into the tables.

The second method for speeding up database access is **using the exists statement in SQL rather than the in statement.** For example

```
Select * from table_a a

Where exists (select * from orders o

where a.prod_id=o.prod_id);
```

is the best way to write an SQL statement with a subquery.

The third method is to **avoid using SQL functions in WHERE clauses or predicate clause**. An expression using a column, for example such as a function having a column as an argument, can cause the SQL optimizer to ignore the use of an index on that column.

This is an example of SQL code that should not be used:

```
 Where to_number (SUBSTR(a.order_no, INSTR(b.order_no, '.') - 1))
= to_number (SUBSTR(a.order_no, INSTR(b.order_no, '.') - 1))
```

Another example of a problem with a function and an index is:

```
Select name

From orders

Where amount !=0;
```

The problem with this query is that an index cannot tell you what is not in the data! So the index is not used in this case.

Change this WHERE clause to

```
Where amount > 0;
```

And the index will be used.

The fourth method is advice to **not use HAVING clauses in select statements.** The reason for this is simple: having only filters rows after all the rows have been returned. In most queries, you do not want all rows returned, just a subset of rows. Therefore, only use HAVING when summary operations are applied to columns restricted by the WHERE clause.

```
Select state from order where state =’NC’;  group by state ;
```

Is much faster than

```
Select state from order group by state having  state =’NC’;
```

Another method is to **minimize the use of subqueries** , instead use join statements when the data is contained in a small number of tables.

Instead of this query:

```
Select ename

From employees emp

where exists  (select price from prices

where prod_id = emp.prod_id and class=’J’);
```

Use this query instead:

```
Select ename ,

From prices pr, employees emp

where pr.prod_id=emp.prod_id and  class=’J’;
```

The order that tables are listed in the SQL statement can greatly impact the speed of a query. In most cases, **the table with the greatest number of rows should be listed first in a query.** The reason is that the SQL parser moves from right to left rather than left to right. It scans the last table listed, and merges all of the rows from the first table with the rows in the last table.

For example, if table Tab1 has 20,000 rows and Tab2 has 1 row then

Select count (*) from Tab1, Tab2 is the best way to write the query

Instead of

```
Select count (*) from Tab2, Tab1
```

When querying data from multiple tables it’s very common to perform a join between the tables. However, a join is not always needed. A very simple way to query two tables with one query is to use the following code.

```
Select A.name, a.grade,
B.name, b.grade
From emp a, empx b
Where b.emp_no=1010 and a.emp_no=2010;
```

When performing a join with distinct, it’s much more efficient to use exists rather than DISTINCT

```
Select date, name
From sales s
Where exists (select ‘X’ from
Employee emp
Where emp.prod_id = s.prod_id);
```

(X is a dummy variable that is needed to make this query work correctly)

```
Select distinct date,name
From sales s,  employee emp
Where s.prod_id=emp.prod_id;
```

EXISTS is a faster alternative because the database realizes that when the sub-query has been satisfied once, the query can be terminated.

The performance of group by queries can be improved by removing unneeded rows early in the selection process. The following queries return the same data. However, the second query is potentially faster, since rows will be removed from the query before the set operators are applied.

```
Select title, avg(pay_rate)

From employees

Group by job

Having job=’Manager’;
```

Is not as good as

```
Select title, avg(pay_rate)

From employees

Having job=’Manager’

Group by job ;
```

SAS MACRO SPEED INCREASES

It’s very common for big data projects that use SAS to employ a lot of SAS macros. Macros save a lot of time in coding, and they also make code much easier to reuse and debug. The downside to macros is that if they are not used correctly, they can actually slow down a program rather than speed it up. This is especially true if some of the macro debugging features are turned on once the code is fully tested and ready to be put into production status. Here are some techniques to use to make sure that macros do not slow down a program.

The basic tip for using macros is that after debugging the macro is complete, set the system options NOMLOGIC, NOMPRINT, NOMRECALL, and NOSYMBOLGEN. If these options are not used for a production job, there are two main problems that can happen First, the log file can grow to be very large. In some programs with a lot of code that loops many times, the log file can grow so large that it can fill up the disk and cause the program to crash. The other problem is that writing out all those log messages can greatly reduce the speed of the program because disk I/O is a very slow process.

The first macro technique is to use **compiled macros.** A compiled macro runs faster because it does not need to be parsed or compiled when the program runs. Macros should not be compiled until they are fully tested and debugged. One caution with compiled macros is that once they are compiled, they cannot be converted back into readable SAS source code. It is essential to store the macro code in a safe place so that it can be modified or added to at a later date. Another advantage of compiled macros is that the code is not visible to the user. This is important if you are giving the code to a customer to use but they should not be allowed to view the source code.

Another way to speed up macros is to **avoid nested macros** where a macro is defined inside another macro. This is not optimal because the inner macro is recompiled every time the outer macro is executed. And when you are processing millions of records, recompiling a macro for each record can really slow down the program. In general, it's also easier to maintain macros that are not nested. It's much better to define two or more macros separately as shown below:

```
%macro m1;

<macro 1 code goes here>

%mend m1;

%macro m2;

<macro 2 code goes here>

%mend m1;
```

Instead of

```
%macro m1;

   %macro m2;

    %mend m2;

%mend m1;
```

Calling a macro from a macro will not slow down the processing, because it will not cause the called macro to be recompiled every time the main macro is called:

```
%macro test1;

%another_macro (this macro was defined outside of macro test1)

%mend test1;
```

Although %include is technically part of the macro language, one big difference is that any code that is put into the program with %include is not compiled as a macro. Therefore, it will run faster than a normal macro. The best use for %include along with macros is to put simple statements in the include file :

```
%let dept_name=Sales;

%let number_div=4;
```

Another good idea for using %include to speed up a program is that an external shell program that calls SAS can write out values as SAS code into a text file, which are then included into the SAS code at the time of execution. This technique allows one SAS program to be written that is very flexible. The way the program is run depends on inputs passed to the code from the external program. The external program can be written in a variety of languages such as C, C++, HTML, or Java. This method also means that the SAS code never has to be changed. Therefore, there is less chance of bugs being introduced into the program. The SAS code can even be stored at read only so that no changes can be made to the source code.

Here is how this method works:

- SAS source code is written with %include statements to subset data
- The external shell program collects information from the end user – for example “Species=Mice”
- The external shell program writes out a line for each piece of information collected - %let species=Mice;   or if species=Mice;
- SAS program is called from the shell program and %include files are executed
- Program runs faster because the correct data subset is used

The external shell program can be simple or complex. The main goal is to collect information to speed up the execution of the SAS program by making sure the correct data is used. The shell program can also collect information from users to use in formatting of tables, colors, output format (such as ODS methods), log location, output location, and so on.

GRID COMPUTING OPTION

The best option to speed up a big data project with SAS is to use grid computing. This option is not inexpensive or simple to set up but for now it is the “ultimate” way to increase computing power and decrease processing time for SAS processing. SSO currently uses a grid system for one of our large retail projects that uses a lot of data, and has a very tight timeline to complete the daily and weekly processing.

The key points for a grid environment at SSO are:

- Windows servers for end-user access
- One SAS server
- One database server
- Four or more grid servers, which are used for the main computing
- RAID disk arrays for fast access to data
- High speed (10 Gb) access between main server and grid nodes
- SAS Grid computing software package

Here is a diagram of a typical grid computing layout used at SSO:

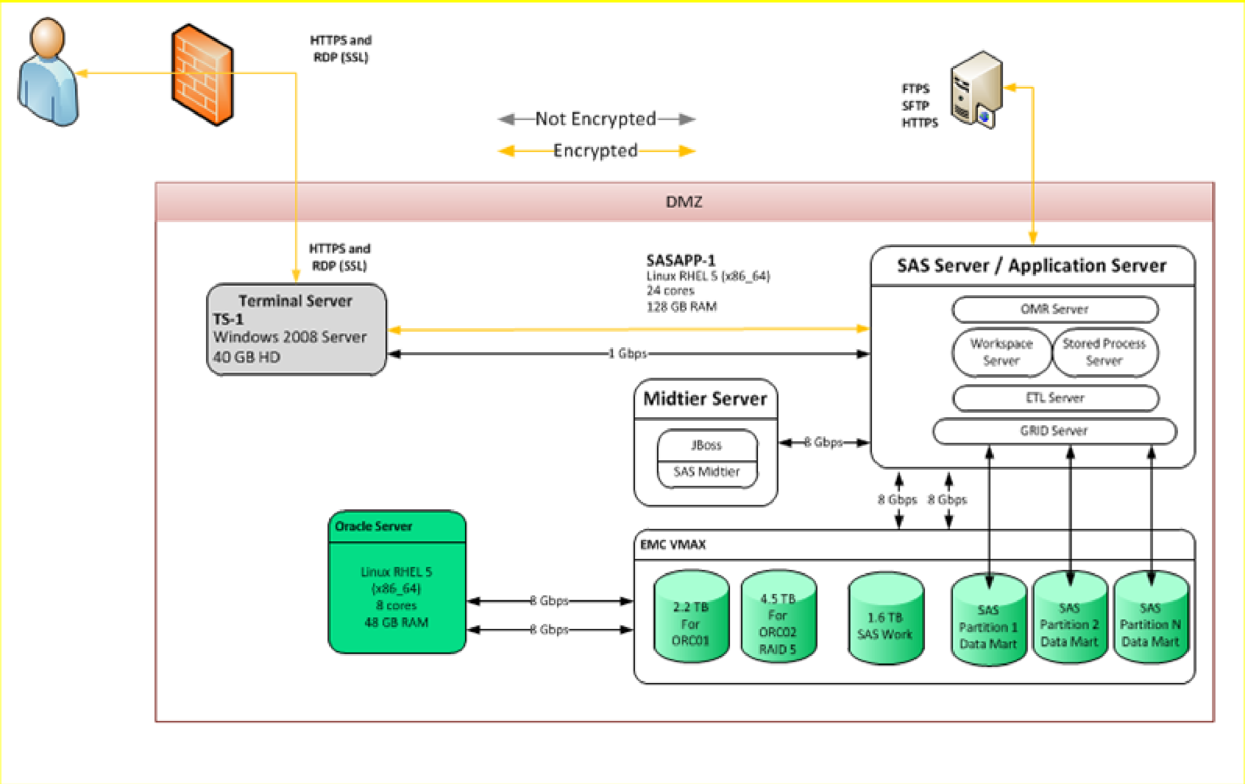

Figure 1 Grid Computing Layout for SSO

In this example, there are 12 grid nodes, each with 12 cores and 64 Gb of RAM (the diagram says 12 of 40 nodes because the other 28 nodes are used for development and test servers.) The "DMZ" means those systems and disk are located together in one location. There is a DMZ for the main SAS server, and an Oracle server and a separate DMZ for the grid nodes.

The key advantages of grid computing are:

- Improved program distribution and CPU utilization

- Can be used for multiple users and multiple applications
- Job, queue, and host management services
- Grid nodes can be setup as hot backups for main server
- Simplifies administration of multiple systems
- Allows easy maintenance since grid nodes can be shut down without disrupting application
- Provides real-time monitoring of systems and applications

In SSO, the grid nodes are not used for data loading (ETL) or reporting. They are used only for the “number crunching” aspect of the project. The normal data flow for a grid computing project in SSO is as follows:

1. Data is loaded daily and weekly using the SAS server and Oracle server
2. Data is partitioned into subsets that match the number of grid nodes (10 sets for 10 grid nodes). The user can select the method for partitioning
3. During processing, the data is copied to the grid nodes for complex calculations
4. When processing is done, the results are copied back to the SAS and Oracle servers

It is technically possible to use a grid computing architecture without using the extra grid node systems. This method still partitions the data, and processes the data in smaller batches. But, it is not as fast as the full grid system shown above.

Of course, it is important to point out the disadvantages of a grid system:

- Greatly increased cost for software and hardware vs. a non-grid system
- More points of potential failure (grid nodes, connections, etc.)
- Harder to set up and maintain a grid vs. a single server system
- A grid might not speed up all processing such as data loads
- Extra CPU processing time is used copying data back and forth to the grid nodes
- More disk I/O is used in a grid system

GENERAL SAS PROGRAMMING IDEAS FOR FASTER PROCESSING OF BIG DATA

The topics in this paper cover what might be considered areas that might not apply to all SAS programs. The rest of this paper will give advice on standard SAS programming – the data step along with various procedures (procs). This is an important area to consider because it is applicable to a wide variety of programs. Not every SAS program will use SQL or macros, but every SAS program will use at least one data step.

The basic idea to speed up processing with the data step is to reduce the amount of work that SAS needs to do. One simple way to do this is to make sure that when possible, a section of code is only executed one time instead of many times (one time for each record in the data.) For example, the easiest way to do this is by using the retain statement.

Another little known method for increasing the speed of calculations in a data step involves the use of missing values. If a variable is known to have a lot of missing values, it is a best practice to list that variable last in a mathematical expression. For example, if the variable T4 has a lot of missing values then

```
Total=(x*b)+c*(abc) + T4;
```

Is more efficient than

```
Total=T4+(x*b)+c*(abc);
```

The reason for this is that if the missing value is early, that missing value is propagated through all the calculations and SAS has to use more CPU time to compute the values and keep track of the missing values. It is also a good idea to check for a missing value first by using code like this:

```
If T4 ne . then do

Total=(x*b)+c*(abc) + T4;

End;
```

In most cases, PROC format is a much faster way to assign values to data rather than using a long list of if then statements.   Statements like this:

```
if educ = 0 then neweduc="< 3 yrs old";
else if educ=1 then neweduc="no school";
else if educ=2 then neweduc="nursery school";
```

Should be converted to this code:

```
proc format;
value educf
0="< 3 yrs old"
1="no school"
2="nursery school";

data new;
set old;
```

```
neweduc=put(educ,educf.);

run;
```

In a similar manner, the use of the in function uses less CPU time than a group of or statements. Instead of

```
If x=8 or x=9 or x=23 or x=45 then do;
```

Use

```
If x in (8,9,23,45) then do;
```

The reason for this change is that with the use of or, SAS checks all the conditions. The in function stops after it finds the first matching number to make the expression true.

SAS uses more CPU time when it has to process larger volumes of data. A very easy way to reduce the size of the data is to avoid using the default data size for variables. By default, all SAS numeric variables have a size of 8 bytes. For many variables, 8 bytes is much larger than is needed. For example, a variable that is used for the age of a person can easily be stored in 3 bytes, which means that the size of the data for that one variable has been reduced by 5/8 or 62.5%. When dealing with very large data sets, that number in the hundreds of millions of records, the CPU processing time savings can be substantial.

Many programs that were written in older versions of SAS can be changed to take advantage of more modern SAS programming features. In older versions of SAS, procedures could not run on a subset of data. If the analysis needed to be run on just one sex for example, a new data set was created that included just members of the sex needed. Now it’s much quicker to use a subset statement in the procedure statements such as

```
Proc freq;

  where sex=’Male’; run;
```

Or

```
Proc means;

 where sex=’Female’; run;
```

In many cases, it is possible to write code using either traditional SAS DATA steps and PROCs or write code using SQL statements in place of the DATA steps and PROCs. These two pieces of code produce the same results:

```
Data abc ;
  Set old_data;
  Keep name date city
Proc sort;
By name, date, city;
```

Vs

```
Proc sql;
Create table abc as
Select name,date ,city
From old_data
Order by name, date, city;
```

One advantage of the SQL code is that it is more compact and easier to read (assuming knowledge of SQL programming.) The question comes up: which method is faster? The SQL code appears to be faster since there is one step versus two steps in the data step code. The truth is there is no easy answer to that question. The answer to which is faster really depends on many different factors:

- Amount of data processed
- How many indexes are used
- Hardware and software configuration (Windows versus Linux or Unix, PC versus Mainframe and so on.)
- Type of analysis needed – can it be done in the database or only in SAS

If possible, it's a good idea to test DATA steps and PROCs versus SQL processing on a small subset of data to determine which method is fastest.

SAS IN-DATABASE PRODUCT

SAS has a relatively new product called SAS In-Database. The big advantage of this product is that is allows SAS jobs to run directly in the database server. Most databases have a very limited feature set for statistical analysis – adding SAS directly into a database greatly increases the amount of analysis that can be done without needing to pull data into SAS (using SQL or a DATA step) and potentially sending the results back to the database. Currently, In-Database works on the following databases: Aster database, EMC Greenplum®, IBM DB2® and Netezza®, Oracle, and Teradata®. In-Database uses massive parallel processing (MPP) to enhance system scalability and performance. It's much better to move the processing to the data rather than move the data to SAS, especially considering the fact that I/O is one of the main factors that can slow down the speed of a program. The three parts of In-Database are:

- SAS Scoring Accelerator
- Analytics Accelerator for Teradata
- SAS Analytic Advantage for Teradata

THREADS AND CPU COUNT OPTIONS

These two options can be very helpful for speeding up processing but it's important to be careful when you are using them. In general, it's best to use them only for very large data sets. Using them on smaller data sets might actually slow down processing. The SAS system will decide if these options are actually used based on different factors such as number of CPUs installed in the system, or options selected for a given DATA step or procedure that is used. It's also a very good idea to test the threads option versus nothreads to make sure that the speed does increase by using threads.

The best way to use the CPU and threads option is:

```
Options threads cpu=actual;
```

The actual statement on the CPU option tells SAS to use the actual number of CPUs installed in the system. It might be tempting to try to use a higher number for CPUs. In reality, it does not work that way. Also, this option means the programmer does not have to spend time learning how many CPUs are in the system.

A simple way to explain the threads option is that it divides the work up into smaller "chunks" so that they can be worked on in parallel by different CPUs. This is very helpful in many different SAS procedures. if the program uses PROC SQL with the pass through option, the threads option will have no impact because the SQL code is passed to the database where it is executed. The pass through option treats the database as a sort of "black box" that SAS has no control over. However, it is possible the database system might use its own version of multithreading to speed up processing within the database.

Another important fact about the threads option is that the results can vary depending on what type of hardware is used to run the SAS program. For example, a program that uses threads on Linux might not work as well if the source code is run on Windows or a mainframe system.

DON'T FORGET ABOUT MAKING OLD/EXISTING CODE RUN FASTER!

Programmers read technical papers such as this one and decide to start using these techniques in the future. While that is a very good idea, all the information in this article can (and should) be used to examine old code to determine if the old code can be improved. Just because old code has been running without problems (sometimes for years) does not mean that the code is efficient. "If it is not broken, don't fix it" is a good saying but sometimes a program might be broken even when it produces the correct results. In this context, "broken" means that the code can be changed to run faster while still producing the correct results.

CONCLUSION

With data volumes increasing all the time, it is important to always be mindful of ways to speed up processing. It can be very tempting to simply "throw money at the problem" by buying faster or bigger servers to run the SAS code. Better hardware can potentially speed up the processing, but it is far from the cheapest way to increase performance. This paper presented two basic ways to decrease processing time for big data projects – by using better programming techniques with SQL, SAS macros, and general SAS programming techniques, and by using multiple servers in a grid environment. The first three methods can be implemented at very low costs, so they should be evaluated for all projects. For organizations with larger budgets or very large amounts of data, the grid environment is a good choice to investigate.

ACKNOWLEDGEMENTS

I would like to thank the writing staff at SAS for editing help on this paper. I would also like to thank all my coworkers at RTI, SRA and SAS who have helped me become a better SAS programmer throughout my career. Special thanks to Dr. Bill Sanders at the University of Tennessee (and later the director of the SAS EVAAS group) who showed me SAS programming for the very first time.

CONTACT INFORMATION

Kevin McGowan
SAS Solutions on Demand
Kevin.McGowan@sas.com
(919) 531-2731
http://www.sas.com