Paper 127-29

Efficiency Techniques for Beginning PROC SQL Users

Kirk Paul Lafler, Software Intelligence Corporation

Abstract

PROC SQL provides SAS users with a powerful programming language that can rival DATA step coding techniques either in terms of the processing time or the human time involved to write the code. Attendees of this hands-on workshop will learn how to use PROC SQL statements, options, and clauses to conserve CPU, I/O, and memory resources while accomplishing tasks involving processing, ordering, grouping, and summarizing data. Numerous hands-on exercises will reinforce the PROC SQL coding techniques that efficiently utilize available resources. No previous experience with PROC SQL is required to attend this workshop.

Introduction

As PROC SQL becomes increasingly more popular, guidelines for its efficient use is critical. Areas deserving special consideration include program execution, I/O, disk space, and program maintenance. A collection of useful techniques and sample code are presented to illustrate numerous practical techniques for gaining efficiency while using PROC SQL.

When developing PROC SQL program code and/or applications, efficiency is not always given the attention it deserves, particularly in the early phases of development. System performance requirements can greatly affect the behavior an application exhibits. Active user participation is crucial to understanding application and performance requirements.

Attention should be given to each individual program function to assess performance criteria. Understanding user expectations, preferably during the early phases of the application development process, can often result in a more efficient application. More often than not though, efficiency techniques are best implemented only after a thorough examination of its performance has been achieved. This provides a greater ability to control what efficiency techniques will best improve the performance of PROC SQL code.

This paper highlights several areas where a program's performance can be improved when using PROC SQL.

Efficiency Objectives

The process of gaining efficiency consists of improving the way PROC SQL code operates. It involves taking program code and exploring what can be done to improve performance in an intelligent, controlled manner. As you might imagine, finely tuned PROC SQL code is code that benefits the most from the existing hardware and software environment.

The process of implementing efficiency techniques involves measuring, evaluating, and modifying PROC SQL code until it uses the minimum amount of computer resources to complete its execution. The biggest problem with the tuning process is that it is sometimes difficult to determine the amount of computer resources a program uses. Complicating matters further, inadequate and incomplete information about resource utilization is often unavailable. In fact, no simple formula exists to determine how efficient a program runs. Often, the only way to assess whether a program is running efficiently is to evaluate its performance under varying conditions, such as during interactive use or during shortages of specific resources including memory and storage.

Efficiency objectives are generally best achieved when implemented as early as possible, preferably during the design or development phase of a program’s life cycle. But when this is not possible, for example when customizing or inheriting an application, efficiency and performance techniques can still be "applied" quite successfully to obtain a significant degree of improvement. Efficiency and performance strategies can be classified into five areas as follows:

1. CPU Time
2. Data Storage
3. I/O
4. Memory
5. Programming Time

Jeffrey A. Polzin of SAS Institute Inc. has this to say about measuring efficiency, *"CPU time and elapsed time are baseline measurements, since all the other measurements impact these in one way or another."* He continues by saying, *"... as one measurement is reduced or increased, it influences the others in varying degrees."*

The simplest of requests can fall prey to one or more efficiency violations, such as retaining unwanted tables in work space, not subsetting early to eliminate undesirable records, or reading wanted as well as unwanted variables. Much of an application's inefficiency can be avoided with better planning and knowing what works and what does not prior to beginning the coding process. Most people do not plan to fail - they just fail to plan. Fortunately, efficiency gains can be realized by following a few guidelines.

Guidelines to Hold Dear

The difference between PROC SQL code that has been optimized versus code that has not can be dramatic. By adhering to a few simple but practical guidelines, an application can achieve greater efficiency in direct relationship to economies of scale. Generally, the first 90% of efficiency improvements are gained relatively quickly and easily by applying simple strategies. It is often the final 10% that, when pursued, proves to be a greater challenge. Consequently, you will need to be the judge of whether your application's code has reached "relative" optimal efficiency while maintaining a virtual balance between time and cost.

There is a general school of thought that says there is no better time than now to deal with efficiency issues. The following checklist shows a successful approach to tune programs.

- Assess load on operating system, usually with the help of a systems administrator.
- Develop list of efficiency objectives.
- Analyze the performance of key program components with the help of users.

The specific areas that should be analyzed include:

- basic algorithms
- macro code
- how data is accessed
- coding techniques

Before implementing one or more efficiency techniques, make copies of any PROC SQL programs to provide a way to recover from inadvertent problems resulting from changes. Then you will be ready to modify any program statements, clauses, and options, if needed. To evaluate the success of any program code modifications, plan to conduct some level of parallel testing.

PROC SQL Coding Examples

The following suggestions are not meant to be an exhaustive review of all known efficiency techniques, but a sampling of proven methods that, when implemented, can provide some measure of improvement in the way PROC SQL code operates. Program examples illustrate the application of a few popular efficiency techniques in the areas of CPU time, data storage, I/O, memory, and programming time.

CPU Time

1) Use the SQL procedure to consolidate the number of steps in an application program. The SQL procedure frequently simplifies and consolidates coding requirements resulting in fewer program steps and reduced CPU requirements. The next example illustrates PROC SQL that consolidates a sort process and data selection in a single step.

   Example:

```
proc sql;
  select *
    from sugi.movies
      order by rating;
quit;
```

2) Use a KEEP= (or DROP=) table option to retain desired table columns. By retaining just the desired columns CPU time is devoted to handling only the data that is wanted, not to unwanted or unneeded data. This essentially reduces CPU-related resources by instructing PROC SQL to process only the specified columns, eliminating any unwanted data from being loaded into memory. Note: Since the KEEP= and DROP= table options do not adhere to ANSI (American National Standards Institute) guidelines, users desiring a level of portability to SQL dialects in other database environments should refrain from using these SAS-specific table options.

Example:

```
proc sql;
  select *
    from sugi.movies (KEEP=title length category);
quit;
```

3) Apply WHERE clause processing when possible to restrict the number of rows in the result table. A WHERE clause restricts the number of rows that will be subset in the result table. This reduces the amount of CPU-related resources that will be expended during program execution.

Example:

```
proc sql;
  select *
    from sugi.movies
      WHERE rating = ‘G’;
quit;
```

4) When constructing a chain of AND-ed conditions in a WHERE clause, specify the most restrictive values first. By constructing AND-ed conditions in this way, CPU resources will be reduced.

Example:

```
proc sql;
  select *
    from sugi.movies
      where rating = ‘G’  AND
            length < 120
        order by title;
quit;
```

5) Avoid unnecessary sorting by planning when an ORDER BY clause is needed. Sorting data in PROC SQL, as in other parts of the SAS System, is a CPU and memory intense operation. When sufficient amounts of CPU and memory resources are available, the process is usually successful. But if either of these resources is in short supply or simply not available, the sort step is most likely doomed for failure. The first order of business for SAS users is to know when an ORDER BY clause is needed and when it is not. When a sort is needed, attempt to sort data one time in the most detailed order as possible. Applying this simple rule will minimize the number of sort requests while reducing CPU-related requirements.

Example:

```
proc sql;
  select *
    from sugi.movies
      ORDER BY category, year;
quit;
```

6) Use the SELECT DISTINCT clause to invoke an internal sort, sometimes referred to as an automatic sort, to remove duplicate rows. In some situations, several rows in a table will contain identical column values. To select only one of each duplicate value, the DISTINCT keyword is used in the SELECT statement as follows.

Example:

```
proc sql;
  select DISTINCT rating
    from sugi.movies;
quit;
```

Output Results:

```
Rating
G
PG
PG-13
R
```

7) When performing a sort-merge join operation, use the SORTEDBY table option to inform the SQL procedure that the table is already arranged in the desired order. This prevents the larger table from being automatically sorted during the join operation.

Example:

```
proc sql;
  select title, rating, length, actor_leading
    from sugi.movies (SORTEDBY=title),
         sugi.actors
      where movies.title = actors.title;
quit;
```

8) Use care when creating indexes by avoiding the temptation of creating too many indexes. Indexes can be used to allow rapid access to table rows. Rather than physically sorting a table (as performed by the ORDER BY clause or PROC SORT), an index is designed to set up a logical arrangement of the data without the need to physically sort it. This has the advantage of reducing CPU requirements.

Indexes require additional CPU-related resources to maintain them. Although very useful, indexes do possess drawbacks. Let's examine what happens when an index is created for a table. As data in a table is inserted, modified, or deleted, an index must be updated to address those changes. This automatic feature requires additional CPU resources to process any changes to a table.

A few simple rules should be kept in mind when considering whether an index is warranted or not. Sequential access to a table may be faster than indexed access when a table is relatively small or when the table is infrequently accessed. An index is useful when created on discriminating columns (or variables) in a table. Avoid creating an index when the page count is less than 3 pages in size for any column in a table (from PROC CONTENTS output). Indexes are most effective when the subset of data based on WHERE clause processing is 25% or less of the population.

Example:

```
proc sql;
  create index rating on sugi.movies;

  select *
    from sugi.movies
      where rating = 'PG';
quit;
```

9) Use CASE logic for data reclassification. In the SQL procedure, a case expression provides a way of conditionally selecting result values from each row in a table. Similar to a DATA step SELECT statement (or IF-THEN/ELSE construct), a case expression uses a WHEN-THEN clause to conditionally process some but not all the rows in a table. An optional ELSE expression can be specified to handle an alternative action should none of the expression(s) identified in the WHEN condition(s) not be satisfied.

In the next example, suppose a value of “All Audiences”, “Adult Supervision”, “Adult”, or “Unknown” is desired for each of the movies. Using the movie’s rating system (RATING) column, a CASE expression can be constructed to assign the desired value in a unique column for each row of data. A column heading of Movie_Type is assigned to the new derived output column using the AS keyword.

Example:

```
proc sql;
  select title,
         rating,
         CASE
           WHEN rating = 'G'     THEN 'All Audiences'
           WHEN rating = 'PG'    THEN 'Adult Supervision'
           WHEN rating = 'PG-13' THEN 'Adult Supervision'
           WHEN rating = 'R'     THEN 'Adult'
           ELSE 'Unknown'
         END AS Movie_Type
    from sugi.movies;
quit;
```

Output Results:

```
                          The SAS System

Title                             Rating  Movie_Type
Brave Heart                       R       Adult
Casablanca                        PG      Adult Supervision
Christmas Vacation                PG-13   Adult Supervision
Coming to America                 R       Adult
Dracula                           R       Adult
Dressed to Kill                   R       Adult
Forrest Gump                      PG-13   Adult Supervision
Ghost                             PG-13   Adult Supervision
Jaws                              PG      Adult Supervision
Jurassic Park                     PG-13   Adult Supervision
Lethal Weapon                     R       Adult
Michael                           PG-13   Adult Supervision
National Lampoon's Vacation       PG-13   Adult Supervision
Poltergeist                       PG      Adult Supervision
Rocky                             PG      Adult Supervision
Scarface                          R       Adult
Silence of the Lambs              R       Adult
Star Wars                         PG      Adult Supervision
The Hunt for Red October          PG      Adult Supervision
The Terminator                    R       Adult
The Wizard of Oz                  G       All Audiences
Titanic                           PG-13   Adult Supervision
```

Data Storage

1) Use KEEP= or DROP= table options to retain desired column variables. Retain only the columns desired at the "read" level in the FROM clause to reduce data storage requirements. For further details, refer to the description of the KEEP= or DROP= table options in the CPU Time section earlier.

2) Use a CREATE TABLE statement only when a table is needed. When a SELECT statement is specified without a CREATE TABLE statement, the result table is an intermediate temporary table residing in memory. The result table only exists for the duration of program execution – and no longer.

3) Compress large tables to reduce data storage requirements. Although compressing a table can be an effective way to reduce the storage requirements for large tables consisting of an abundance of repetitive data such as blanks, it does require additional CPU resources to compress and uncompress tables. Note: Compressing a small table may not provide any data storage improvements. In fact, compressing a small table may achieve negative results by increasing the size required to store a table, see SAS Log Results below.

Example:

```
proc sql;
  create table sugi.pg_movies (COMPRESS=YES) as
    select *
      from sugi.movies
        where rating in ('PG', 'PG-13');
quit;
```

SAS Log Results:

```
1    proc sql;
2      create table sugi.pg_movies (COMPRESS=YES) as
3        select *
4          from sugi.movies
5            where rating in ('PG', 'PG-13');
NOTE: Compressing data set SUGI.PG_MOVIES increased size by 100.00 percent.
      Compressed is 2 pages; un-compressed would require 1 pages.
NOTE: Table SUGI.PG_MOVIES created, with 13 rows and 6 columns.
6    quit;
```

I/O

1) Use KEEP= or DROP= table options to retain desired column variables. Retain only the columns desired at the "read" level in the FROM clause or on the CREATE TABLE statement to reduce I/O requirements. Essentially, when fewer columns are read or written, I/O requirements are reduced. For further details, refer to the description of the KEEP= or DROP= table options in the CPU Time section discussed earlier.

2) Apply WHERE clause processing when possible to restrict the number of rows in the result table. A WHERE clause restricts the number of rows that will be subset in the result table. Because there are fewer rows, this reduces the amount of I/O-related resources that will be expended during program execution. For further details, refer to the description of the WHERE clause processing in the CPU Time section discussed earlier.

3) Use DICTIONARY Tables to access system-related SAS session detail information. SAS users can quickly and conveniently obtain useful information about their SAS session with a number of read-only SAS data views called DICTIONARY tables. At any time during a SAS session, DICTIONARY tables can be used to capture information related to currently defined libnames, table names, column names and attributes, formats, and much more.

DICTIONARY tables are accessed using the libref DICTIONARY in the FROM clause of a PROC SQL SELECT statement. By accessing the information in one or more DICTIONARY tables, costly CPU and I/O resources are eliminated or reduced. The next example illustrates how the number of rows in the MOVIES table can be retrieved by accessing DICTIONARY.TABLES in the FROM clause of a SELECT statement.

Example:

```
proc sql;
  select LIBNAME, MEMNAME, NOBS
    from DICTIONARY.TABLES
      where upcase(libname) = 'SUGI' and
            upcase(memname) = 'MOVIES' and
            upcase(memtype) = 'DATA';
quit;
```

Output Results:

```
                         The SAS System

                                                   Number of
    Library                                         Physical
    Name      Member Name                       Observations
    ----------------------------------------------------------
    SUGI      MOVIES                                      23
```

Memory

1) Use KEEP= or DROP= table options to retain desired column variables. Retain only the columns desired at the "read" level in the FROM clause to reduce memory requirements. Essentially, when fewer columns are retained, memory requirements are reduced. For further details, refer to the description of the KEEP= or DROP= table options in the CPU Time section discussed earlier.

Programming Time

1) Use the SQL procedure for code simplification, fewer program steps, greater portability, and more maintainable program code. The SQL procedure frequently simplifies and consolidates coding requirements resulting in fewer program steps and reduced programming time requirements. For further details, refer to the SQL procedure description in the CPU Time section discussed earlier.

2) Store formats with the SAS tables that use them. By storing formats in a SAS table, programming time is reduced because the selected column will display the data using the predefined format. This not only saves time, it enables output to be generated quickly and easily without having to define formats each time a table is accessed.

Example:

```
proc sql;
  alter table sugi.rental_info
    modify rental_date FORMAT=MMDDYY10.;

  select *
    from sugi.rental_info;
quit;
```

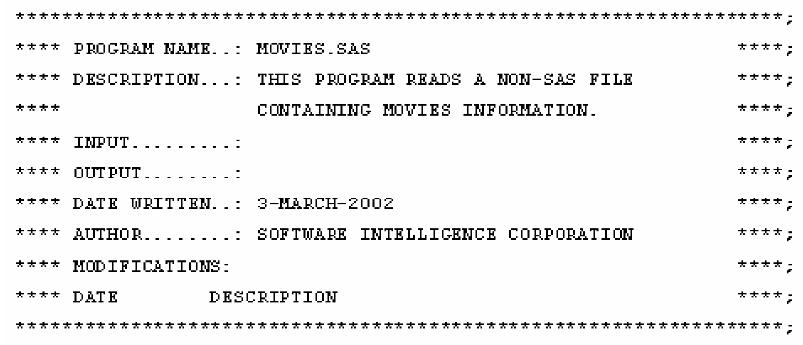
Output Results:

```
                             The SAS System

TITLE                                RENTAL_AMT  RENTAL_DATE
Brave Heart                               $3.99   04/20/2004
Star Wars                                 $3.99   05/01/2004
The Wizard of Oz                          $2.50   03/28/2004
```

3) Document programs and routines with comments to minimize maintenance time and reduce the learning curve related to complex code logic.

Example:



```
*******************************************************************;
**** PROGRAM NAME..: MOVIES.SAS                               ****;
**** DESCRIPTION...: THIS PROGRAM READS A NON-SAS FILE        ****;
****                 CONTAINING MOVIES INFORMATION.           ****;
**** INPUT.........:                                          ****;
**** OUTPUT........:                                          ****;
**** DATE WRITTEN..: 3-MARCH-2002                             ****;
**** AUTHOR........: SOFTWARE INTELLIGENCE CORPORATION        ****;
**** MODIFICATIONS:                                           ****;
**** DATE        DESCRIPTION                                  ****;
*******************************************************************;
```

```
proc sql;
  select *
    from sugi.movies

      < other SQL code >;

quit;
```

4) Code for unknown data values using CASE logic. This will prevent unassigned or null data values from falling through logic conditions. For further details, refer to the SQL procedure description in the CPU Time section discussed earlier.

5) Assign descriptive and meaningful variable names. Esoteric abbreviations and naming conventions provide little help to users who are expected to understand the inner complexities of a program. Provide meaningful names to variables.

Learning Necessary Techniques

So how do people learn about efficiency techniques? A small number learn through formal training. Others find published guidelines (e.g., book(s), manuals, articles, etc.) on the subject. The majority indicated they learn techniques as a result of a combination of prior experiences, through acquaintances (e.g., User Groups), and/or on the job. Any improvement is better than no improvement. Consequently, adhering to a practical set of guidelines can benefit significantly for many years to come.

1) An inadequate level of formal training exists on efficiency and performance techniques.

2) A failure to plan in advance of the coding phase.

3) Insufficient time and inadequate budgets can often be attributed to ineffective planning and implementation of efficiency strategies.

Conclusion

The value of implementing efficiency and performance strategies into PROC SQL code cannot be over-emphasized. Careful attention should be given to individual program functions, since one or more efficiency techniques can often affect the architectural characteristics and/or behavior applications exhibit.

Efficiency techniques are learned in a variety of ways. Many users learn valuable techniques through formal classroom instruction, while others find value in published guidelines such as books, manuals, articles, and “white” papers. But the greatest value comes from other’s experiences, as well as their own, by word-of-mouth, and on the job. Whatever the means, a little efficiency goes along way.

Acknowledgments

The author would like to thank the Hands-on Workshops Section Chairs Deb Cassidy and Ginger Carey for selecting this topic, for their hard work, and for inviting me as a speaker in the Hands-on Workshops Section at SUGI 29. I would also like to thank Duke Owen, SUGI 29 Conference Chair for all his hard work and for doing a great job as SUGI 29 Conference Chair. Thank you!

References

Lafler, Kirk Paul (2003), “Undocumented and Hard-to-find PROC SQL Features”, Proceedings of the Eleventh Annual Western Users of SAS Software (WUSS) Conference.

Lafler, Kirk Paul, SAS® Fundamentals, Version 8 Course Notes, Revised and Updated (2001), Software Intelligence Corporation, Spring Valley, CA, USA.

Lafler, Kirk Paul (2000), "Efficient SAS Programming Techniques", MWSUG 2000 Conference.

Lafler, Kirk Paul (1996), "Gaining Efficiency with SAS Software," Proceedings of the Twenty-First Annual SAS Users Group International (SUGI) Conference.

Polzin, Jeffrey A. (1994), "DATA Step Efficiency and Performance", Proceedings of the Nineteenth Annual SAS Users Group International Conference, 1574-1580.

Wilson, Steven A. (1994), "Techniques for Efficiently Accessing and Managing Data", Proceedings of the Nineteenth Annual SAS Users Group International Conference, 207-212.

Hardy, Jean E. (1992), "Efficient SAS Software Programming: A Version 6 Update", Proceedings of the Seventeenth Annual SAS Users Group International Conference, 207-212.

Fournier, Roger (1991), Practical Guide to Structured System Development and Maintenance, Yourdon Press Series. Englewood Cliffs, N.J.: Prentice-Hall, Inc., 136-143.

Valentine-Query, Paige (1991), "Introduction to Efficient Programming Techniques", Proceedings of the Sixteenth Annual SAS Users Group International Conference, 266-270.

SAS Institute Inc. (1990), SAS Programming Tips: A Guide to Efficient SAS Processing, Cary, NC, USA.

Lafler, Kirk Paul (1985), "Optimization Techniques for SAS Applications", Proceedings of the Tenth Annual SAS Users Group International Conference, 530-532.

Trademark Citations

About the Author

Kirk Paul Lafler, a SAS Certified Professional® and former SAS Alliance Partner® (1996 – 2002), with more than 25 years of SAS software experience provides consulting services and hands-on SAS training around the world. Kirk has written four books including Power SAS and Power AOL by Apress and more than one hundred articles for professional journals and SAS User Group proceedings. His popular SAS Tips column, Kirk's Korner, appears regularly in the BASAS, HASUG, SANDS, and SESUG Newsletters. His expertise includes application design and development, training, and programming using Base SAS, PROC SQL, ODS, SAS/FSP, SAS/AF, SCL, and FRAME software.

Comments and suggestions can be sent to:

Kirk Paul Lafler
Author, Trainer, and Consultant
Software Intelligence Corporation
P.O. Box 1390
Spring Valley, California 91979-1390
E-mail: KirkLafler@cs.com
Voice: 619.660.2400