Paper 114-31

A Hands-On Tour Inside the World of PROC SQL

Kirk Paul Lafler, Software Intelligence Corporation

Abstract

Structured Query Language (PROC SQL) is a database language found in the base-SAS software. It enables access to data stored in SAS data sets or tables using a powerful assortment of statements, clauses, options, functions, and other language features. This hands-on workshop presents core concepts of this powerful language as well as its many applications, and is intended for SAS users who desire an overview of the capabilities of this exciting procedure. Attendees will explore the construction of simple SQL queries, ordering and grouping data, the application of case logic for data reclassification, the creation and use of views, and the construction of simple inner and outer joins.

Introduction

The SQL procedure is a wonderful tool for querying and subsetting data; restructuring data by constructing case expressions; constructing and using virtual tables known as view; joining two or more tables (up to 32) to explore data relationships. Occasionally, a problem comes along where the SQL procedure is either better suited or easier to use than other more conventional DATA and/or PROC step methods. As each situation presents itself, PROC SQL should be examined to see if its use is warranted for the task at hand.

PROC SQL proponents frequently use the power and coding simplicity of the SQL procedure to perform an assortment of common everyday tasks to those requiring a highly complex and analytical approach. Whatever the nature of SQL usage, this paper is intended to present the five most exciting features found in PROC SQL.

Example Tables

The examples used throughout this paper utilize a database of tables. A relational database is a collection of tables. Each table contains one or more columns and one or more rows of data. The example database consists of three tables: CUSTOMERS, MOVIES, and ACTORS. Each table appears below.

CUSTOMERS

CUST_NO	NAME	CITY	STATE
11321	John Smith	Miami	FL
44555	Alice Jones	Baltimore	MD
21713	Ryan Adams	Atlanta	GA

MOVIES

CUST_NO	MOVIE_NO	RATING	CATEGORY
44555	1011	PG-13	Adventure
21713	3090	G	Comedy
44555	2198	G	Comedy
37753	4456	PG	Suspense

ACTORS

MOVIE_NO	LEADING_ACTOR
1011	Mel Gibson
2198	Clint Eastwood
3090	Sylvester Stallone

Constructing SQL Queries to Retrieve and Subset Data

PROC SQL provides simple, but powerful, retrieval and subsetting capabilities. From inserting a blank row between each row of output, removing rows with duplicate values, using wildcard characters to search for partially known information, and integrating ODS with SQL to create nicer-looking output.

Inserting a Blank Row into Output

SQL can be made to automatically insert a blank row between each row of output. This is generally a handy feature when a single logical row of data spans two or more lines of output. By having SQL insert a blank row between each logical record (observation), you create a visual separation between the rows – making it easier for the person reading the output to read and comprehend the output. The DOUBLE option is specified as part of the SQL procedure statement to insert a blank row and is illustrated in the following SQL code.

SQL Code

```
PROC SQL DOUBLE;
 SELECT *
  FROM MOVIES
   ORDER BY category;
QUIT;
```

Removing Rows with Duplicate Values

When the same value is contained in several rows in a table, SQL can remove the rows with duplicate values. By specifying the DISTINCT keyword prior to the column that is being selected in the SELECT statement automatically removes duplicate rows as illustrated in the following SQL code.

SQL Code

```
PROC SQL;
 SELECT DISTINCT rating
  FROM MOVIES;
QUIT;
```

Using Wildcard Characters for Searching

When searching for specific rows of data is necessary, but only part of the data you are searching for is known, then SQL provides the ability to use wildcard characters as part of the search argument. Say you wanted to search for all movies that were classified as an “Action” type of movie. By specifying a query using the wildcard character percent sign (%) in a WHERE clause with the LIKE operator, the query results will consist of all rows containing the word “ACTION” as follows.

SQL Code

```
PROC SQL;
 SELECT movie_no, category
  FROM MOVIES
   WHERE UPCASE(category) LIKE ‘%ACTION%’;
QUIT;
```

Phonetic Matching (Sounds-Like Operator =)*

A technique for finding names that sound alike or have spelling variations is available in the SQL procedure. This frequently used technique is referred to as phonetic matching and is performed using the Soundex algorithm. In Joe Celko’s book, SQL for Smarties: Advanced SQL Programming, he traced the origins of the Soundex algorithm to the developers Margaret O’Dell and Robert C. Russell in 1918.

Although not technically a function, the sounds-like operator searches and selects character data based on two expressions: the search value and the matched value. Anyone that has looked for a last name in a local telephone directory is quickly reminded of the possible phonetic variations.

To illustrate how the sounds-like operator works, let's search each movie title for the phonetic variation of "Suspence" which, by the way, is spelled incorrectly. To help find as many (and hopefully all) possible spelling variations, the sounds-like operator is used to identify select similar sounding names including spelling variations. To find all movies where the movie title sounds like "Suspence", the following code is used:

SQL Code

```
PROC SQL;
 SELECT movie_no, category, rating
  FROM MOVIES
   WHERE category =* 'Suspence';
QUIT;
```

Case Logic

In the SQL procedure, a case expression provides a way of conditionally selecting result values from each row in a table (or view). Similar to an IF-THEN 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.

A case expression must be a valid SQL expression and conform to syntax rules similar to DATA step SELECT-WHEN statements. Even though this topic is best explained by example, let's take a quick look at the syntax.

```
CASE <column-name>
      WHEN when-condition THEN result-expression
     <WHEN when-condition THEN result-expression> …

     <ELSE result-expression>
END
```

A column-name can optionally be specified as part of the CASE-expression. If present, it is automatically made available to each when-condition. When it is not specified, the column-name must be coded in each when-condition. Let's examine how a case expression works.

If a when-condition is satisfied by a row in a table (or view), then it is considered "true" and the result-expression following the THEN keyword is processed. The remaining WHEN conditions in the CASE expression are skipped. If a when-condition is "false", the next when-condition is evaluated. SQL evaluates each when-condition until a "true" condition is found or in the event all when-conditions are "false", it then executes the ELSE expression and assigns its value to the CASE expression's result. A missing value is assigned to a CASE expression when an ELSE expression is not specified and each when-condition is "false".

In the next example, let's see how a case expression actually works. Suppose a value of "Exciting", "Fun", "Scary", or " " is desired for each of the movies. Using the movie's category (CATEGORY) column, a CASE expression is constructed to assign one of the desired values in a unique column called TYPE for each row of data. A value of 'Exciting' is assigned to all Adventure movies, 'Fun' for Comedies, 'Scary' for Suspense movies, and blank for all other movies. A column heading of TYPE is assigned to the new derived output column using the AS keyword.

SQL Code

```
PROC SQL;
  SELECT MOVIE_NO,
         RATING,
         CASE
           WHEN CATEGORY = 'Adventure' THEN 'Exciting'
            WHEN CATEGORY = 'Comedy'    THEN 'Fun'
            WHEN CATEGORY = 'Suspense'  THEN 'Scary'
            ELSE ''
         END AS TYPE
    FROM MOVIES;
QUIT;
```

In another example suppose we wanted to determine the audience level (general or adult audiences) for each movie. By using the RATING column we can assign a descriptive value with a simple Case expression, as follows.

SQL Code

```
PROC SQL;
  SELECT MOVIE_NO,
         RATING,
         CASE RATING
           WHEN ‘G’ THEN ‘General’
           ELSE ‘Other’
         END AS Audience_Level
    FROM MOVIES;
QUIT;
```

Creating and Using Views

Views are classified as virtual tables. There are many reasons for constructing and using views. A few of the more common reasons are presented below.

Minimizing, or perhaps eliminating, the need to know the table or tables underlying structure

Often a great degree of knowledge is required to correctly identify and construct the particular table interactions that are necessary to satisfy a requirement. When this prerequisite knowledge is not present, a view becomes a very attractive alternative. Once a view is constructed, users can simply execute it. This results in the underlying table(s) being processed. As a result, data integrity and control is maintained since a common set of instructions is used.

Reducing the amount of typing for longer requests

Often, a query will involve many lines of instruction combined with logical and comparison operators. When this occurs, there is any number of places where a typographical error or inadvertent use of a comparison operator may present an incorrect picture of your data. The construction of a view is advantageous in these circumstances, since a simple call to a view virtually eliminates the problems resulting from a lot of typing.

Hiding SQL language syntax and processing complexities from users

When users are unfamiliar with the SQL language, the construction techniques of views, or processing complexities related to table operations, they only need to execute the desired view using simple calls. This simplifies the process and enables users to perform simple to complex operations with custom-built views.

Providing security to sensitive parts of a table

Security measures can be realized by designing and constructing views designating what pieces of a table's information is available for viewing. Since data should always be protected from unauthorized use, views can provide some level of protection (also consider and use security measures at the operating system level).

Controlling change / customization independence

Occasionally, table and/or process changes may be necessary. When this happens, it is advantageous to make it as painless for users as possible. When properly designed and constructed, a view modifies the underlying data without the slightest hint or impact to users, with the one exception that results and/or output may appear differently. Consequently, views can be made to maintain a greater level of change independence.

Types of Views

Views can be typed or categorized according to their purpose and construction method. Joe Celko, author of SQL for Smarties and a number of other SQL books, describes views this way, *"Views can be classified by the type of SELECT statement they use and the purpose they are meant to serve."* To classify views in the SAS System environment, one must also look at how the SELECT statement is constructed. The following classifications are useful when describing a view's capabilities.

Single-Table Views

Views constructed from a single table are often used to control or limit what is accessible from that table. These views generally limit what columns, rows, and/ or both are viewed.

Ordered Views

Views constructed with an ORDER BY clause arrange one or more rows of data in some desired way.

Grouped Views

Views constructed with a GROUP BY clause divide a table into sets for conducting data analysis. Grouped views are more often than not used in conjunction with aggregate functions (see aggregate views below).

Distinct Views

Views constructed with the DISTINCT keyword tell the SAS System how to handle duplicate rows in a table.

Aggregate Views

Views constructed using aggregate and statistical functions tell the SAS System what rows in a table you want summary values for.

Joined-Table Views

Views constructed from a join on two or more tables use a connecting column to match or compare values. Consequently, data can be retrieved and manipulated to assist in data analysis.

Nested Views

Views can be constructed from other views, although extreme care should be taken to build views from base tables.

Creating Views

When creating a view, its name must be unique and follow SAS naming conventions. Also, a view cannot reference itself since it does not already exist. The next example illustrates the process of creating an SQL view. In the first step, no output is produced since the view must first be created. Once the view has been created, the second step executes the view, G_MOVIES, by rendering the view's instructions to produce the desired output results.

SQL Code

```
PROC SQL;
 CREATE VIEW G_MOVIES AS
  SELECT movie_no, category, rating
   FROM MOVIES
    WHERE rating = ‘G’
     ORDER BY movie_no;
 SELECT *
  FROM G_MOVIES;
QUIT;
```

Exploring Indexes

Indexes can be used to improve the access speed to desired table rows. Rather than physically sorting a table (as performed by the ORDER BY clause or the BY statement in PROC SORT), an index is designed to set up a logical data arrangement without the need to physically sort it. This has the advantage of reducing CPU and memory requirements. It also reduces data access time when using WHERE clause processing.

Understanding Indexes

What exactly is an index? An index consists of one or more columns in a table to uniquely identify each row of data within the table. Operating as a SAS object containing the values in one or more columns in a table, an index is comprised of one or more columns and may be defined as numeric, character, or a combination of both. Although there is no rule that says a table must have an index, when present, they are most frequently used to make information retrieval using a WHERE clause more efficient.

To help determine when an index is necessary, it is important to look at existing data as well as the way the base table(s) will be used. It is also critical to know what queries will be used and how they will access columns of data. There are times when the column(s) making up an index are obvious and other times when they are not. When determining whether an index provides any processing value, some very important rules should be kept in mind. An index should permit the greatest flexibility so every column in a table can be accessed and displayed. Indexes should also be assigned to discriminating column(s) only since query results will benefit greatest when this is the case.

When an index is specified on one or more tables, a join process may actually be boosted. The PROC SQL processor may use an index, when certain conditions permit its use. Here are a few things to keep in mind before creating an index:

- If the table is small, sequential processing may be just as fast, or faster, than processing with an index
- If the page count as displayed in the CONTENTS procedure is less than 3 pages, avoid creating an index
- Do not to create more indexes than you absolutely need
- If the data subset for the index is not small, sequential access may be more efficient than using the index
- If the percentage of matches is approximately 15% or less then an index should be used
- The costs associated with an index can outweigh its performance value – an index is updated each time the rows in a table are added, deleted, or modified.

Sample code will be illustrated below on creating simple and composite indexes using the CREATE INDEX statement in the SQL procedure.

Creating a Simple Index

A simple index is specifically defined for one column in a table and must be the **same** name as the column. Suppose you had to create an index consisting of movie rating (RATING) in the MOVIES table. Once created, the index becomes a separate object located in the SAS library.

SQL Code

```
    PROC SQL;
      CREATE INDEX RATING ON MOVIES(RATING);
    QUIT;
```

SAS Log Results

```
    PROC SQL;
      CREATE INDEX RATING ON MOVIES(RATING);
NOTE: Simple index RATING has been defined.
     QUIT;
```

Creating a Composite Index

A composite index is defined for two or more columns in a table and must have a **unique** name that is different than the columns assigned to the index. Suppose you were to create an index consisting of movie rating (RATING) and movie category (CATEGORY) in the MOVIES table. Once the composite index is created, the index consisting of the two table columns become a separate object located in the SAS library.

SQL Code

```
    PROC SQL;
      CREATE INDEX RATING_CAT ON MOVIES(RATING, CATEGORY);
    QUIT;
```

SAS Log Results

```
    PROC SQL;
      CREATE INDEX RATING_CAT ON MOVIES(RATING, CATEGORY);
NOTE: Composite index RATING_CAT has been defined.
    QUIT;
```

Index Limitations
Indexes can be very useful, but they do have limitations. As data in a table is inserted, modified, or deleted, an index must also be updated to address any and all changes. This automatic feature requires additional CPU resources to process any changes to a table. Also, as a separate structure in its own right, an index can consume considerable storage space. As a consequence, care should be exercised not to create too many indexes but assign indexes to only those discriminating variables in a table.

Because of the aforementioned drawbacks, indexes should only be created on tables where query search time needs to be optimized. Any unnecessary indexes may force the SAS System to expend resources needlessly updating and reorganizing after insert, delete, and update operations are performed. And even worse, the PROC SQL optimizer may accidentally use an index when it should not.

Optimizing WHERE Clause Processing with Indexes
A WHERE clause defines the logical conditions that control which rows a SELECT statement will select, a DELETE statement will delete, or an UPDATE statement will update. This powerful, but optional, clause permits SAS users to test and evaluate conditions as true or false. From a programming perspective, the evaluation of a condition determines which of the alternate paths a program will follow. Conditional logic in PROC SQL is frequently implemented in a WHERE clause to reference constants and relationships among columns and values.

To get the best possible performance from programs containing SQL procedure code, an index and WHERE clause can be used together. Using a WHERE clause restricts processing in a table to a subset of selected rows. When an index exists, the SQL processor determines whether to take advantage of it during WHERE clause processing. Although the SQL processor determines whether using an index will ultimately benefit performance, when it does the result can be an improvement in processing speeds.

PROC SQL Joins
A join of two or more tables provides a means of gathering and manipulating data in a single SELECT statement. A "JOIN" statement does not exist in the SQL language. The way two or more tables are joined is to specify the tables names in a WHERE clause of a SELECT statement. A comma separates each table specified in an inner join.

Joins are specified on a minimum of two tables at a time, where a column from each table is used for the purpose of connecting the two tables. Connecting columns should have ***"like"*** values and the same datatype attributes since the join's success is dependent on these values.

What Happens During a Join?
Joins are processed in two distinct phases. The first phase determines whether a FROM clause references two or more tables. When it does, a *virtual* table, known as the Cartesian product, is created resulting in each row in the first table being combined with each row in the second table, and so forth. The Cartesian product (internal virtual table) can be extremely large since it represents all possible combinations of rows and columns in the joined tables. As a result, the Cartesian product can be, and often is, extremely large.

The second phase of every join processes, if present, is specified in the WHERE clause. When a WHERE clause is specified, the number of rows in the Cartesian product is reduced to satisfy this expression. This data subsetting process generally results in a more manageable end product containing meaningful data.

Creating a Cartesian Product
When a WHERE clause is omitted, all possible combinations of rows from each table is produced. This form of join is known as the ***Cartesian Product***. Say for example you join two tables with the first table consisting of 10 rows and the second table with 5 rows. The result of these two tables would consist of 50 rows. Very rarely is there a need to perform a join operation in SQL where a WHERE clause is not specified. The primary importance of this form of join is to illustrate a base for all joins. As illustrated in the following diagram, the two tables are combined without a corresponding WHERE clause. Consequently, no connection between common columns exists.

CUSTOMERS
Cust_no
Name
City
State

MOVIES
Cust_no
Movie_no
Rating
Category

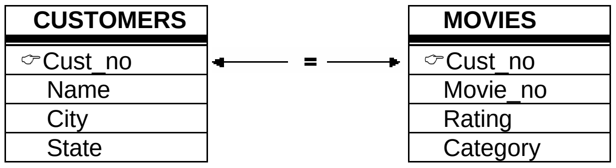
The result of a Cartesian Product is the combination of all rows and columns. The next example illustrates a Cartesian Product join using a SELECT query without a WHERE clause.

SQL Code

```
PROC SQL;
  SELECT *
    FROM CUSTOMERS, MOVIES;
QUIT;
```

Joining Two Tables with a Where Clause

Joining two tables together is a relatively easy process in SQL. To illustrate how a join works, a two-table join is linked using the customer number (CUST_NO) in the following diagram.



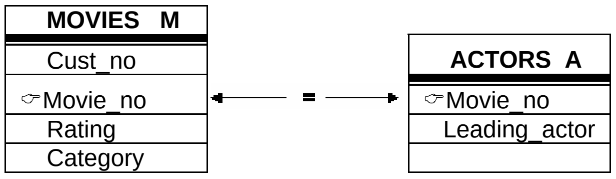
The next SQL code references a join on two tables with CUST_NO specified as the connecting column. In this example, tables CUSTOMERS and MOVIES are used. Each table has a common column, CUST_NO which is used to connect rows together from each when the value of CUST_NO is equal, as specified in the WHERE clause. A WHERE clause restricts what rows of data will be included in the resulting join.

SQL Code

```
PROC SQL;
  SELECT *
    FROM CUSTOMERS, MOVIES
      WHERE CUSTOMERS.CUST_NO  =  MOVIES.CUST_NO;
QUIT;
```

Joins and Table Aliases

Table aliases provide a "short-cut" way to reference one or more tables in a join operation. One or more aliases are specified so columns can be selected with a minimal number of keystrokes. To illustrate how table aliases in a join works, a two-table join is linked in the following diagram.



The following SQL code illustrates a join on two tables with MOVIE_NO specified as the connecting column. The table aliases are specified in the SELECT statement as qualified names, the FROM clause, and the WHERE clause.

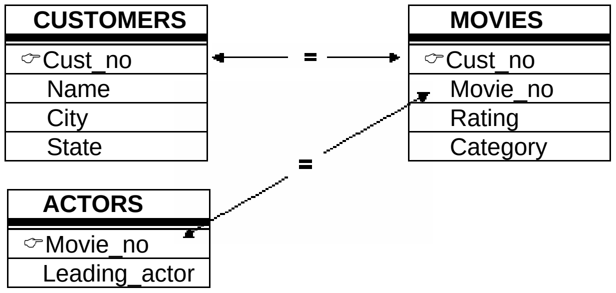
SQL Code

```
PROC SQL;
  SELECT M.MOVIE_NO,
         M.RATING,
         A.LEADING_ACTOR
    FROM MOVIES M, ACTORS A
      WHERE M.MOVIE_NO  =  A.MOVIE_NO;
QUIT;
```

Joining Three Tables

In an earlier example, you saw where customer information was combined with the movies they rented. You may also want to display the leading actor of each movie along with the other information. To do this, you will need to extract information from three different tables: CUSTOMERS, MOVIES, and ACTORS.

A join with three tables follows the same rules as in a two-table join. Each table will need to be listed in the FROM clause with appropriate restrictions specified in the WHERE clause. To illustrate how a three table join works, the following diagram should be visualized.



The next SQL code example references a join on three tables with CUST_NO specified as the connecting column for the CUSTOMERS and MOVIES tables, and MOVIE_NO as the connecting column for the MOVIES and ACTORS tables.

<u>**SQL Code**</u>

```
PROC SQL;
  SELECT C.CUST_NO,
         M.MOVIE_NO,
         M.RATING,
         M.CATEGORY,
         A.LEADING_ACTOR
    FROM CUSTOMERS C,
         MOVIES M,
         ACTORS A
      WHERE C.CUST_NO = M.CUST_NO  AND  M.MOVIE_NO = A.MOVIE_NO;
QUIT;
```

Introduction to Outer Joins

Generally a join is a process of relating rows in one table with rows in another. But occasionally, you may want to include rows from one or both tables that have no related rows. This concept is referred to as row preservation and is a significant feature offered by the outer join construct.

There are operational and syntax differences between inner (natural) and outer joins. First, the maximum number of tables that can be specified in an outer join is two (the maximum number of tables that can be specified in an inner join is 32). Like an inner join, an outer join relates rows in both tables. But this is where the similarities end because the result table also includes rows with no related rows from one or both of the tables. This special handling of “matched” and “unmatched” rows of data is what differentiates an outer join from an inner join.

An outer join can accomplish a variety of tasks that would require a great deal of effort using other methods. This is not to say that a process similar to an outer join can not be programmed – it would probably just require more work. Let’s take a look at a few tasks that are possible with outer joins:

- List all customer accounts with rentals during the month, including customer accounts with no purchase activity.
- Compute the number of rentals placed by each customer, including customers who have not rented.
- Identify movie renters who rented a movie last month, and those who did not.

Another obvious difference between an outer and inner join is the way the syntax is constructed. Outer joins use keywords such as LEFT JOIN, RIGHT JOIN, and FULL JOIN, and has the WHERE clause replaced with an ON clause. These distinctions help identify outer joins from inner joins.

Finally, specifying a left or right outer join is a matter of choice. Simply put, the only difference between a left and right join is the order of the tables they use to relate rows of data. As such, you can use the two types of outer joins interchangeably and is one based on convenience.

Exploring Outer Joins

Outer joins process data relationships from two tables differently than inner joins. In this section a different type of join, known as an outer join, will be illustrated. The following code example illustrates a left outer join to identify and match movie numbers from the MOVIES and ACTORS tables. The resulting output would contain all rows for which the SQL expression, referenced in the ON clause, matches both tables and all rows from the left table (MOVIES) that did not match any row in the right (ACTORS) table. Essentially the rows from the left table are preserved and captured exactly as they are stored in the table itself, regardless if a match exists.

SQL Code

```
PROC SQL;
 SELECT movies.movie_no, leading_actor, rating
  FROM MOVIES
      LEFT JOIN
       ACTORS
    ON movies.movie_no = actors.movie_no;
QUIT;
```

The next example illustrates the result of using a right outer join to identify and match movie titles from the MOVIES and ACTORS tables. The resulting output would contain all rows for which the SQL expression, referenced in the ON clause, matches in both tables (is true) and all rows from the right table (ACTORS) that did not match any row in the left (MOVIES) table.

SQL Code

```
PROC SQL;
 SELECT movies.movie_no, actor_leading, rating
  FROM MOVIES
     RIGHT JOIN
       ACTORS
    ON movies.movie_no = actors.movie_no;
QUIT;
```

Conclusion

The SQL procedure is a wonderful tool for SAS users to explore and use in a variety of application situations. This paper has presented a few of most exciting features found in PROC SQL. You are encouraged to explore PROC SQL's powerful capabilities as it relates to querying and subsetting data; restructuring data by constructing case expressions; constructing and using virtual tables known as view; joining two or more tables (up to 32) to explore data relationships.

References

Lafler, Kirk Paul (2004). *PROC SQL: Beyond the Basics Using SAS*, SAS Institute Inc., Cary, NC, USA.

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

Lafler, Kirk Paul (1992-2006). *PROC SQL for Beginners*; Software Intelligence Corporation, Spring Valley, CA, USA.

Lafler, Kirk Paul (1998-2006). *Intermediate PROC SQL*; Software Intelligence Corporation, Spring Valley, CA, USA.

SAS® Guide to the SQL Procedure: Usage and Reference, Version 6, First Edition (1990). SAS Institute, Cary, NC.

SAS® SQL Procedure User's Guide, Version 8 (2000). SAS Institute Inc., Cary, NC, USA.

Trademarks

Acknowledgments

I would like to thank Lori Griffin and Derek Morgan, SUGI Hands-on Workshops Section Co-Chairs, for accepting my abstract and paper, as well as the SUGI Leadership for their support of a great Conference.

Bio

Kirk Paul Lafler, a SAS Certified Professional® and SAS Alliance Partner® (1996 - 2002), has used SAS software since 1979. As a consultant and trainer with Software Intelligence Corporation, he provides consulting services and hands-on SAS training to users around the world. Kirk has written four books including PROC SQL: Beyond the Basics Using SAS by SAS Institute, Power SAS and Power AOL by Apress, and more than one hundred peer-reviewed articles and papers for professional journals and SAS User Group proceedings. His popular SAS Tips column, Kirk's Korner, appears regularly in the BASAS, HASUG, SANDS, SAS, SESUG, and WUSS Newsletters and websites. Kirk can be reached at:

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