Undocumented and Hard-to-find SQL Features

Kirk Paul Lafler, Software Intelligence Corporation

Abstract

The SQL Procedure contains many powerful and elegant language features for advanced SQL users. This paper presents SQL topics that will help programmers unlock the many hidden features, options, and other hard-to-find gems found in the SQL universe. Topics include CASE logic; the COALESCE function; SQL statement options _METHOD, _TREE, and other useful options; dictionary tables; automatic macro variables; and performance issues.

Finding the First Non-Missing Value

The SQL procedure provides a way to find the first non-missing value in a column or list. Specified in a SELECT statement, the COALESCE function inspects a column, or in the case of a list scans the arguments from left to right, and returns the first non-missing or non-NULL value. If all values are missing, the result is missing.

When coding the COALESCE function, all arguments must be of the same data type. The example shows one approach on computing the total number of minutes in the MOVIES table. In the event either the LENGTH or RATING columns contain a missing value, a zero is assigned to prevent the propagation of missing values.

PROC SQL Code

```
PROC SQL;
  SELECT TITLE, RATING, (COALESCE(LENGTH, 0)) AS Tot_Length
    FROM MOVIES;
QUIT;
```

Results

```
                The SAS System

Title                    Rating Tot_Length
──────────────────────────────────────────
Brave Heart              R             177
Casablanca               PG            103
Christmas Vacation       PG-13          97
Coming to America        R             116
Dracula                  R             130
Dressed to Kill          R             105
Forrest Gump             PG-13         142
Ghost                    PG-13         127
Jaws                     PG            125
Jurassic Park            PG-13         127
Lethal Weapon            R             110
Michael                  PG-13         106
National Lampoon's Vacat PG-13          98
Poltergeist              PG            115
Rocky                    PG            120
Scarface                 R             170
Silence of the Lambs     R             118
Star Wars                PG            124
The Hunt for Red October PG            135
The Terminator           R             108
The Wizard of Oz         G             101
Titanic                  PG-13         194
```

Summarizing data

Although the SQL procedure is frequently used to display or extract detailed information from tables in a database, it is also a wonderful tool for summarizing (or aggregating) data. By constructing simple queries, data can be summarized down rows (observations) as well as across columns (variables). This flexibility gives SAS• users an incredible range of power, and the ability to take advantage of several SAS-supplied (or built-in) summary functions. For example, it may be more interesting to see the average of some quantities rather than the set of all quantities.

Without the ability to summarize data in SQL, users would be forced to write complicated formulas and/or routines, or even write and test DATA step programs to summarize data. To see how an SQL query can be constructed to summarize data, two examples will be illustrated: 1) Summarizing data down rows and 2) Summarizing data across rows.

1. Summarizing data down rows

The first example shows a single aggregate result value being produced when movie-related data is summarized down rows (or observations). The advantages of using a summary function in SQL is that it will generally compute the aggregate quicker than if a user-defined equation were constructed and it saves the effort of having to construct and test a program containing the user-defined equation in the first place. Suppose you wanted to know the average length of all PG and PG-13 movies in a database table containing a variety of movie categories. The following query computes the average movie length and produces a single aggregate value using the AVG function.

PROC SQL Code

```
PROC SQL;
 SELECT AVG(LENGTH) AS Average_Movie_Length
  FROM MOVIES
   WHERE RATING IN (“PG”, “PG-13”);
QUIT;
```

The result from executing this query shows that the average movie length rounded to the hundredths position is 124.08 minutes.

Results

```
   Average_
Movie_Length
   124.0769
```

2. Summarizing data across columns

Being able to summarize data across columns often comes in handy, when a computation is required on two or more columns in each row. Suppose you wanted to know the difference in minutes between each PG and PG-13 movie’s running length with trailers (add-on specials for your viewing pleasure) and without trailers.

PROC SQL Code

```
PROC SQL;
 SELECT TITLE, RANGE(LENGTH_TRAIL, LENGTH) AS Extra_Minutes
  FROM MOVIES
   WHERE RATING IN (“PG”, “PG-13”);
QUIT;
```

This query computes the difference between the length of the movie and its trailer in minutes and once computed displays the range value for each row as Extra_Minutes.

Results

Title	___	Extra_ Minutes
Casablanca		0
Jaws		0
Rocky		0
Star Wars		0
Poltergeist		0
The Hunt for Red October		15
National Lampoon's Vacation		7
Christmas Vacation		6
Ghost		0
Jurassic Park		33
Forrest Gump		0
Michael		0
Titanic		36

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 "Short", "Medium", or "Long" is desired for each of the movies. Using the movie's length (LENGTH) column, a CASE expression is constructed to assign one of the desired values in a unique column called M_Length for each row of data. A value of 'Short' is assigned to the movies that are shorter than 120 minutes long, 'Long' for movies longer than 160 minutes long, and 'Medium' for all other movies. A column heading of M_Length is assigned to the new derived output column using the AS keyword.

PROC SQL Code

```
PROC SQL;
  SELECT TITLE,
         LENGTH,
         CASE
           WHEN LENGTH < 120 THEN 'Short'
           WHEN LENGTH > 160 THEN 'Long'
           ELSE 'Medium'
         END AS M_Length
    FROM MOVIES;
QUIT;
```

Results

The SAS System

Title	Length	M_Length
Brave Heart	177	Long
Casablanca	103	Short
Christmas Vacation	97	Short
Coming to America	116	Short
Dracula	130	Medium
Dressed to Kill	105	Short
Forrest Gump	142	Medium
Ghost	127	Medium
Jaws	125	Medium
Jurassic Park	127	Medium
Lethal Weapon	110	Short
Michael	106	Short
National Lampoon's Vacation	98	Short
Poltergeist	115	Short
Rocky	120	Medium
Scarface	170	Long
Silence of the Lambs	118	Short
Star Wars	124	Medium
The Hunt for Red October	135	Medium
The Terminator	108	Short
The Wizard of Oz	101	Short
Titanic	194	Long

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.

PROC SQL Code

```
PROC SQL;
  SELECT TITLE,
         RATING,
         CASE RATING
           WHEN 'G' THEN 'General'
           ELSE 'Other'
         END AS Aud_Level
    FROM MOVIES;
QUIT;
```

Results

The SAS System

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

PROC SQL and the Macro Language

Many software vendors' SQL implementation permits SQL to be interfaced with a host language. The SAS System's SQL implementation is no different. The SAS Macro Language lets you customize the way the SAS software behaves, and in particular extend the capabilities of the SQL procedure. SQL users can apply the macro facility's many powerful features by interfacing PROC SQL with the macro language to provide a wealth of programming opportunities.

From creating and using user-defined macro variables and automatic (SAS-supplied) variables, reducing redundant code, performing common and repetitive tasks, to building powerful and simple macro applications, SQL can be integrated with the macro language to improve programmer efficiency. The best part is that you do not have to be a macro language heavyweight to begin reaping the rewards of this versatile interface between two powerful Base-SAS software languages.

Creating a Macro Variable with Aggregate Functions

Turning data into information, and then saving the results as macro variables is easy with summary (aggregate) functions. The SQL procedure provides a number of useful summary functions to help perform calculations, descriptive statistics, and other aggregating computations in a SELECT statement or HAVING clause. These functions are designed to summarize information and not display detail about data. In the next example, the MIN summary function is used to determine the least expensive product from the PRODUCTS table with the value stored in the macro variable MIN_PRODCOST using the INTO clause. The results are displayed on the SAS log.

PROC SQL Code

```
PROC SQL NOPRINT;
  SELECT MIN(LENGTH)
    INTO :MIN_LENGTH
      FROM MOVIES;
QUIT;
%PUT &MIN_LENGTH;
```

SAS Log Results

```
PROC SQL NOPRINT;
  SELECT MIN(LENGTH)
    INTO :MIN_LENGTH
      FROM MOVIES;
QUIT;
NOTE: PROCEDURE SQL used:
      real time           0.00 seconds

%PUT &MIN_LENGTH;
97
```

Building Macro Tools

The Macro Facility, combined with the capabilities of the SQL procedure, enables the creation of versatile macro tools and general-purpose applications. A principle design goal when developing user-written macros should be that they are useful and simple to use. A macro that violates this tenant of little applicability to user needs, or with complicated and hard to remember macro variable names, are usually avoided.

As tools, macros should be designed to serve the needs of as many users as possible. They should contain no ambiguities, consist of distinctive macro variable names, avoid the possibility of naming conflicts between macro variables and data set variables, and not try to do too many things. This utilitarian approach to macro design helps gain the widespread approval and acceptance by users.

Column cross-reference listings come in handy when you need to quickly identify all the SAS library data sets a column is defined in. Using the COLUMNS dictionary table a macro can be created that captures column-level information including column name, type, length, position, label, format, informat, indexes, as well as a cross-reference listing containing the location of a column within a designated SAS library. In the next example, macro COLUMNS consists of an SQL query that accesses any single column in a SAS library. If the macro was invoked with a user-request consisting of %COLUMNS(PATH,TITLE);, the macro would produce a cross-reference listing on the user library PATH for the column TITLE in all DATA types.

PROC SQL Code

```
%MACRO COLUMNS(LIB, COLNAME);
PROC SQL;
 SELECT LIBNAME, MEMNAME
    FROM DICTIONARY.COLUMNS
      WHERE UPCASE(LIBNAME)="&LIB" AND
            UPCASE(NAME)="&COLNAME" AND
            UPCASE(MEMTYPE)="DATA";
QUIT;
%MEND COLUMNS;

%COLUMNS(PATH,TITLE);
```

Results

```
The SAS System

Library
Name      Member Name
_____________________________________
PATH      ACTORS
PATH      MOVIES
```

Submitting a Macro and SQL Code with a Function Key

For interactive users using the SAS Display Manager System, a macro can be submitted with a function key. This simple, but effective, technique makes it easy to run a macro with the touch of a key anytime and as often as you like. All you need to do is define the macro containing the instructions you would like to have it perform, include the macro in each session you want to use it in, and enter the SUBMIT command as part of each macro statement to execute the macro. Then, define the desired function key by opening the KEYS window, add the macro name, and save. Anytime you want to execute the macro, simply press the function key you designated.

Suppose you wanted to determine the values of all automatic variables set during the current session. In the next example, you enter and save the following macro statement to inspect the values of current automatic variable settings. By pressing the designated function key, the macro is submitted, executed, and the results displayed.

PROC SQL Code

```
SUBMIT "%PUT _AUTOMATIC_;";
```

Debugging SQL Processing

The SQL procedure offers a couple new options in the debugging process. Two options of critical importance are _METHOD and _TREE. By specifying a _**METHOD** option on the SQL statement, it displays the hierarchy of processing that occurs. Results are displayed on the Log using a variety of codes (see table).

Codes	Description
sqxcrta	Create table as Select
Sqxslct	Select
sqxjsl	Step loop join (Cartesian)
sqxjm	Merge join
sqxjndx	Index join
sqxjhsh	Hash join
sqxsort	Sort
sqxsrc	Source rows from table
sqxfil	Filter rows
sqxsumg	Summary stats with GROUP BY
sqxsumn	Summary stats with no GROUP BY

In the next example a _METHOD option is specified to show the processing hierarchy in a two-way equi-join.

PROC SQL Code

```
PROC SQL _METHOD;
  SELECT MOVIES.TITLE, RATING, ACTOR_LEADING
    FROM MOVIES, ACTORS
      WHERE MOVIES.TITLE = ACTORS.TITLE;
QUIT;
```

Results

```
NOTE: SQL execution methods chosen are:
      sqxslct
          sqxjhsh
              sqxsrc( MOVIES )
              sqxsrc( ACTORS )
```

Another option that is useful for debugging purposes is the _TREE option. In the next example the SQL statements are transformed into an internal form showing a hierarchical layout with objects and a variety of symbols. Inspecting the tree output can frequently provide a greater level of understanding of what happens during SQL processing.

PROC SQL Code

```
PROC SQL  _TREE;
  SELECT  MOVIES.TITLE, RATING, ACTOR_LEADING
    FROM MOVIES, ACTORS
      WHERE MOVIES.TITLE = ACTORS.TITLE;
QUIT;
```

Results

```
NOTE: SQL execution methods chosen are:
sqxslct
       sqxjhsh
          sqxsrc( MOVIES )
          sqxsrc( .ACTORS )
Tree as planned.
                        /-SYM-V-(MOVIES.Title:1 flag=0001)
                 /-OBJ----|
                |       |--SYM-V-(MOVIES.Rating:6 flag=0001)
                |       |--SYM-V-(MOVIES.Length:2 flag=0001)
                |       \-SYM-V-(ACTORS.Actor_Leading:2 flag=0001)
        /-JOIN---|
       |        |                        /-SYM-V-(MOVIES.Title:1 flag=0001)
       |        |                 /-OBJ----|
       |        |                |       |--SYM-V-(MOVIES.Rating:6 flag=0001)
       |        |                |       \-SYM-V-(MOVIES.Length:2 flag=0001)
       |        |       /-SRC----|
       |        |      |       \-TABL[WORK].MOVIES opt=''
       |       |--FROM---|
       |        |      |                /-SYM-V-(ACTORS.Title:1 flag=0001)
       |        |      |       /-OBJ----|
       |        |      |      |       \-SYM-V-(ACTORS.Actor_Leading:2 flag=0001)
       |        |       \-SRC----|
       |        |                \-TABL[WORK].ACTORS opt=''
       |       |--empty-
       |        |       /-SYM-V-(MOVIES.Title:1)
       |        \-CEQ----|
       |                 \-SYM-V-(ACTORS.Title:1)
 --SSEL---|
```

If you have surplus virtual memory, you can achieve faster access to matching rows from one or more small input data sets. Referred to as a **Hash** join the **BUFFERSIZE=** option can be used to let the SQL procedure hash join larger tables. The default BUFFERSIZE=n option is 64000 when not specified. In the next example, a BUFFERSIZE=256000 is specified to utilize available memory to load rows. The result is faster performance because of a hash join.

PROC SQL Code

```
PROC SQL  _method  BUFFERSIZE=256000;
   SELECT MOVIES.TITLE, RATING, ACTOR_LEADING
      FROM MOVIES, ACTORS
         WHERE MOVIES.TITLE = ACTORS.TITLE;
QUIT;
```

Results

```
NOTE: SQL execution methods chosen are:
      sqxslct
          sqxjhsh
              sqxsrc( MOVIES )
              sqxsrc( ACTORS )
```

Conclusion

The SQL Procedure contains many undocumented and hard-to-find powerful and elegant language features for SQL users. This paper highlighted many topics that will help unlock the many hidden features, options, and other hard-to-find gems found in PROC SQL. Topics included CASE logic; the COALESCE function; SQL statement options _METHOD, _TREE, and other useful options; dictionary tables; automatic macro variables; and performance issues.

Acknowledgments

I would like to thank Keith Cranford Conference Chair for accepting my abstract and paper, as well as the SCSUG Leadership for their support of a great Conference.

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 (1992-2004). *PROC SQL for Beginners*; Software Intelligence Corporation, Spring Valley, CA, USA.

Lafler, Kirk Paul (1998-2004). *Intermediate PROC SQL*; Software Intelligence Corporation, Spring Valley, CA, USA.

Lafler, Kirk Paul (2001-2004). *Advanced PROC SQL*; Software Intelligence Corporation, Spring Valley, CA, USA.

Lafler, Kirk Paul (2002). *PROC SQL Programming Tips*; 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, USA.

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

Contact Information

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 PROC SQL: Beyond the Basics Using SAS by SAS Institute (available October 2004), Power SAS and Power AOL by Apress, and more than one hundred articles in professional journals and SAS User Group proceedings. His popular SAS Tips column appears regularly in the BASAS, HASUG, SANDS, SAS, and SESUG Newsletters and websites. Kirk can be reached at:

Kirk Paul Lafler
Software Intelligence Corporation
P.O. Box 1390
Spring Valley, California 91979-1390
Voice: 619-277-7350
E-mail: KirkLafler@cs.com
Web: www.software-intel.com