Powerful and Hard-to-find PROC SQL Features

Kirk Paul Lafler, Software Intelligence Corporation, Spring Valley, California

Abstract

The SQL procedure contains many powerful and elegant language features for intermediate and advanced SQL users. This presentation discusses topics that will help SAS® users unlock the many powerful features, options, and other gems found in the SQL universe. Topics include CASE logic; a sampling of summary (statistical) functions; dictionary tables; PROC SQL and the SAS macro language interface; joins and join algorithms; PROC SQL statement options _METHOD, MAGIC=101, MAGIC=102, and MAGIC=103; and "key" performance (optimization) issues.

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 a view; access information from Dictionary tables; and joining two or more tables to explore data relationships. Occasionally, an application 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.

Example Tables

The examples used throughout this paper utilize a database of two tables. (A relational database is a collection of tables.) The data used in all the examples in this paper consists of a selection of movies that I've viewed over the years. The Movies table consists of six columns: title, length, category, year, studio, and rating. Title, category, studio, and rating are defined as character columns with length and year being defined as numeric columns. The data stored in the Movies table is depicted below.

MOVIES Table

	Title	Length	Category	Year	Studio	Rating
1	Brave Heart	177	Action Adventure	1995	Paramount Pictures	R
2	Casablanca	103	Drama	1942	MGM / UA	PG
3	Christmas Vacation	97	Comedy	1989	Warner Brothers	PG-13
4	Coming to America	116	Comedy	1988	Paramount Pictures	R
5	Dracula	130	Horror	1993	Columbia TriStar	R
6	Dressed to Kill	105	Drama Mysteries	1980	Filmways Pictures	R
7	Forrest Gump	142	Drama	1994	Paramount Pictures	PG-13
8	Ghost	127	Drama Romance	1990	Paramount Pictures	PG-13
9	Jaws	125	Action Adventure	1975	Universal Studios	PG
10	Jurassic Park	127	Action	1993	Universal Pictures	PG-13
11	Lethal Weapon	110	Action Cops & Robber	1987	Warner Brothers	R
12	Michael	106	Drama	1997	Warner Brothers	PG-13
13	National Lampoon's Vacation	98	Comedy	1983	Warner Brothers	PG-13
14	Poltergeist	115	Horror	1982	MGM / UA	PG
15	Rocky	120	Action Adventure	1976	MGM / UA	PG
16	Scarface	170	Action Cops & Robber	1983	Universal Studios	R
17	Silence of the Lambs	118	Drama Suspense	1991	Orion	R
18	Star Wars	124	Action Sci-Fi	1977	Lucas Film Ltd	PG
19	The Hunt for Red October	135	Action Adventure	1989	Paramount Pictures	PG
20	The Terminator	108	Action Sci-Fi	1984	Live Entertainment	R
21	The Wizard of Oz	101	Adventure	1939	MGM / UA	G
22	Titanic	194	Drama Romance	1997	Paramount Pictures	PG-13

The data stored in the ACTORS table consists of three columns: title, actor_leading, and actor_supporting, all of which are defined as character columns. The data stored in the Actors table is illustrated below.

ACTORS Table

	Title	Actor_Leading	Actor_Supporting
1	Brave Heart	Mel Gibson	Sophie Marceau
2	Christmas Vacation	Chevy Chase	Beverly D'Angelo
3	Coming to America	Eddie Murphy	Arsenio Hall
4	Forrest Gump	Tom Hanks	Sally Field
5	Ghost	Patrick Swayze	Demi Moore
6	Lethal Weapon	Mel Gibson	Danny Glover
7	Michael	John Travolta	Andie MacDowell
8	National Lampoon's Vacation	Chevy Chase	Beverly D'Angelo
9	Rocky	Sylvester Stallone	Talia Shire
10	Silence of the Lambs	Anthony Hopkins	Jodie Foster
11	The Hunt for Red October	Sean Connery	Alec Baldwin
12	The Terminator	Arnold Schwarzenegge	Michael Biehn
13	Titanic	Leonardo DiCaprio	Kate Winslet

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 title, 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 title, 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 TITLE, 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 TITLE, 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 leading 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 inform the SAS System the table rows you want summary values computed 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 title, category, rating
      FROM MOVIES
        WHERE rating = 'G'
          ORDER BY movie_no;
 SELECT *
   FROM G_MOVIES;
QUIT;
```

Exploring Dictionary Tables

The SAS System generates and maintains valuable information at run time about SAS libraries, data sets, catalogs, indexes, macros, system options, titles, and views in a collection of read-only tables called dictionary tables. Although called tables, Dictionary tables are not real tables. Information is automatically generated at runtime and each table's contents are made available once a SAS session is started.

The contents from Dictionary tables permit a SAS session's activities to be easily accessed and monitored through the construction of simple queries. This becomes particularly useful in the design and construction of software applications since the information can be queried and the results acted upon in a specific task such as in the allocation of filerefs or librefs.

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. The name of each DICTIONARY table and view along with its purpose are presented below.

DICTIONARY Tables and Purpose

DICTIONARY table	Purpose
CATALOGS	Provides information about SAS catalogs.
CHECK_CONSTRAINTS	Provides check constraints information.
COLUMNS	Provides information about column in tables.
CONSTRAINT_COLUMN_USAGE	Provides column integrity constraints information.
CONSTRAINT_TABLE_USAGE	Provides information related to tables with integrity constraints defined.
DICTIONARIES	Provides information about all the DICTIONARY tables.
EXTFILES	Provides information related to external files.
FORMATS	Provides information related to defined formats and informats.
GOPTIONS	Provides information about currently defined SAS/GRAPH software graphics options.
INDEXES	Provides information related to defined indexes.
LIBNAMES	Provides information related to defined SAS data libraries.
MACROS	Provides information related to any defined macros.
MEMBERS	Provides information related to objects currently defined in SAS data libraries.
OPTIONS	Provides information related to SAS system options.
REFERENTIAL_CONSTRAINTS	Provides information related to tables with referential constraints.
STYLES	Provides information related to select ODS styles.
TABLE_CONSTRAINTS	Provides information related to tables containing integrity constraints.
TABLES	Provides information related to currently defined tables.
TITLES	Provides information related to currently defined titles and footnotes.
VIEWS	Provides information related to currently defined data views.

Displaying Dictionary Table Definitions

A dictionary table's definition can be displayed by specifying a DESCRIBE TABLE statement. The results of the statements and clauses used to create each dictionary table can be displayed on the SAS Log. For example, a DESCRIBE TABLE statement is illustrated below to display the CREATE TABLE statement used in building the OPTIONS dictionary table containing current SAS System option settings.

SQL Code

```
PROC SQL;
 DESCRIBE TABLE
   DICTIONARY.OPTIONS;
QUIT;
```

SAS Log Results

```
create table DICTIONARY.OPTIONS
  (
   optname char(32) label='Option Name',
   setting char(1024) label='Option Setting',
   optdesc char(160) label='Option Description',
   level char(8) label='Option Location'
  );
```

Note: The information contained in dictionary tables is also available to DATA and PROC steps outside the SQL procedure. Referred to as SASHELP views, each view is prefaced with the letter "V". SASHELP views can be accessed by referencing the view by its name in the SASHELP library. Please refer to the SAS Procedures Guide for further details on accessing and using dictionary views in the SASHELP library.

Dictionary.COLUMNS

Retrieving information about the columns in one or more data sets is easy with the COLUMNS dictionary table. Similar to the results of the CONTENTS procedure, you will be able to capture column-level information including column name, type, length, position, label, format, informat, and indexes, as well as produce cross-reference listings containing the location of columns in a SAS library. For example, the following code requests a cross-reference listing of the tables containing the TITLE column in the WORK library.

Note: Care should be used when specifying multiple functions on the WHERE clause since the SQL Optimizer is unable to optimize the query resulting in all allocated SAS session librefs being searched. This can cause the query to run much longer than expected.

SQL Code

```
PROC SQL;
 SELECT *
   FROM DICTIONARY.COLUMNS
    WHERE UPCASE(LIBNAME)="WORK" AND
       UPCASE(NAME)="TITLE";
QUIT;
```

Results

Library Name	Member Name	Member Type	Column Name	Column Type	Column Length	Column Position	Column Number in Table	Column Label	Column Format	Column Informat	Column Index Type
Order in Key Sequence	Extended Type	Not NULL?	Precision	Scale	Transcoded?						
WORK	ACTORS	DATA	Title	char	30	0	1				
0	char	no		.	.	yes					
WORK	MOVIES	DATA	Title	char	30	7	1				SIMPLE
0	char	no		.	.	yes					

When you need more information about SAS files consider using the TABLES dictionary table. The TABLES dictionary table provides detailed information about the library name, member name and type, date created and last modified, number of observations, observation length, number of variables, password protection, compression, encryption, number of pages, reuse space, buffer size, number of deleted observations, type of indexes, and requirements vector. For example, to obtain a detailed list of files in the WORK library, the following code is specified.

Note: Because the TABLE Dictionary table produces a considerable amount of information, users should consider specifying a WHERE clause when using.

SQL Code

```
PROC SQL;
  SELECT *
    FROM DICTIONARY.TABLES
      WHERE UPCASE(LIBNAME)="WORK";
QUIT;
```

Results

Library Name	Member Name	Member Type	DBMS Member Type	Dataset Label	Dataset Type	Date Created	Date Modified	Number of Physical Observations
Observation Length	Number of Variables	Type of Password Protection	Compression Routine	Encryption	Number of Pages	Size of File	Percent Compression	Reuse Space	Bufsize
Number of Deleted Observations	Number of Logical Observations	Longest variable name	Longest label	Maximum number of generations	Generation number	Dataset Attributes	Type of Indexes	Data Representation
Name of Collating Sequence	Sorting Type	Charset Sorted By	Requirements Vector	Data Representation Name	Data Encoding	Audit Trail Active?
Audit Before Image?	Audit Admin Image?	Audit Error Image?	Audit Data Image?

WORK	ACTORS	DATA			DATA	09AUG04:15:40:18	09AUG04:15:40:18	13
70	3	---	NO	NO	1	16384	0	no	8192
0	13	16	0	0	.	ON		NATIVE	
		181F101122220032220102320432012222003E0000100301			WINDOWS_32	wlatin1 Western (Windows)	no
no	no	no	no

WORK	MOVIES	DATA			DATA	09AUG04:15:40:18	09AUG04:15:40:18	22
88	6	---	NO	NO	2	24576	0	no	8192
0	22	8	0	0	.	ON	SIMPLE	NATIVE
		181F101122220032220102320432012222003E0000100301			WINDOWS_32	wlatin1 Western (Windows)	no
no	no	no	no

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. Users can apply the macro facility's many powerful features using the interface between the two languages 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 of this 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.

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 library WORK for the column TITLE in all DATA types.

SQL and Macro 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(WORK,TITLE);
```

Results

```
                         The SAS System


              Library
              Name        Member Name
              WORK        ACTORS
              WORK        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, assign and save the macro call to the desired function key in the KEYS window one time, and include the macro in each session you want to use it in. From that point on, anytime you want to execute the macro, simply press the designated function key.

For example, a simple PROC SQL query can be embedded inside a macro. You will not only save keystrokes by not having to enter it over and over again, but you will improve your productivity as well. The following code illustrates a PROC SQL query embedded within a macro that accesses the "read-only" table DICTIONARY.TABLES. The purpose of the macro and PROC SQL code is to display a "snapshot" of the number of rows in each table that is currently available to the SAS System. Once the macro is defined, it can be called by entering %NOBS on any DMS command line to activate the commands.

SQL and Macro Code

```
%MACRO nobs;
   SUBMIT "PROC SQL; SELECT libname, memname, nobs FROM DICTIONARY.TABLES; QUIT;";
%MEND nobs;
```

To further reduce keystrokes and enhance user productivity even further, a call to a defined macro can be saved to a Function Key. The purpose for doing this would be to allow for one-button operation of any defined macro. To illustrate the process of saving a macro call to a Function Key, the %NOBS macro defined previously is assigned to Function Key F12 in the KEYS window. Once the %NOBS macro call is assigned in the KEYS window, you will be able to call the macro simply by pressing the F12 function key. The partial KEYS window is displayed below to illustrate the process.

KEYS Window

Key	Definition
F1	help
F2	reshow
F3	end;
...	...
F10	keys
F11	command focus
F12	**%NOBS**

Partial Output from Calling %NOBS

```
                         The SAS System

                                              Number of
              Library                          Physical
              Name      Member Name          Observations
              WORK      ACTORS                         13
              WORK      CUSTOMERS                        3
              WORK      MOVIES                          22
              WORK      PG_RATED_MOVIES                 13
              WORK      RENTAL_INFO                     11
```

PROC SQL Join Algorithms

Several PROC SQL join algorithms are available to the SQL optimizer. Based on the constructed query and the underlying table structures, the SQL optimizer determines which of the four available join algorithms to use in executing the join.

- ✓ **Nested Loop** – A nested loop join algorithm may be selected by the SQL optimizer when processing small tables of data where one table is considerably smaller than the other table, the join condition does not contain an equality condition, first row matching is optimized, or using a sort-merge or hash join has been eliminated.

- ✓ **Sort-Merge** – A sort-merge join algorithm may be selected by the SQL optimizer when the tables are small to medium size and an index or hash join algorithm have been eliminated from consideration.

- ✓ **Index** – An index join algorithm may be selected by the SQL optimizer when indexes created on each of the columns participating in the join relationship will improve performance.

- ✓ **Hash** – A hash join algorithm may be selected by the SQL optimizer when sufficient memory is available to the system, and the BUFFERSIZE option is large enough to store the smaller of the tables into memory.

Influencing the SQL Optimizer with Magic

The SQL procedure supports various options to influence the execution of specific join algorithms. The following SQL procedure options are available:

Option	Description
MAGIC=101	Influences the SQL optimizer to select the Nested Loop join algorithm.
MAGIC=102	Influences the SQL optimizer to select the Sort-Merge join algorithm.
MAGIC=103	Influences the SQL optimizer to select the Hash join algorithm.

In the next example, the tables CUSTOMERS and MOVIES are specified in an equijoin construct, as illustrated below. Each table has a common column, CUST_NO which connects rows together when the value of CUST_NO is equal, and is a way to restrict what rows will be included in the join results.

MOVIES		ACTORS
✓ Title	=	✓ Title
Length		Actor_Leading
Category		Actor_Supporting
Rating		
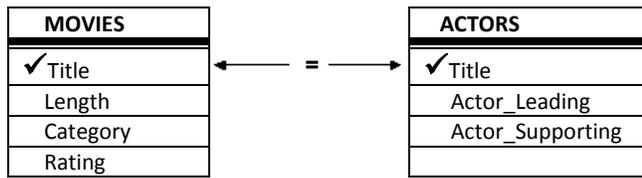
Specifying MAGIC=101

The following SQL procedure code and corresponding SAS Log shows the MAGIC=101 option as the option of choice to influence the optimizer in selecting a nested loop join algorithm for executing the join query.

SQL Code

```
PROC SQL MAGIC=101;
 SELECT  *  FROM  MOVIES, ACTORS
    WHERE  MOVIES.TITLE  =  ACTORS.TITLE;
QUIT;
```

Log Results

```
PROC SQL MAGIC=101;
  SELECT  *  FROM  MOVIES, ACTORS
     WHERE MOVIES.TITLE  =  ACTORS.TITLE;
NOTE: PROC SQL planner chooses sequential loop join.
QUIT;
NOTE: PROCEDURE SQL used (Total process time):
    real time         0.02 seconds
    cpu time          0.01 seconds
```

Specifying MAGIC=102

The following SQL procedure code and corresponding SAS Log shows the MAGIC=102 option being specified to influence the optimizer in selecting a sort-merge join algorithm for executing the join query.

SQL Code

```
PROC SQL MAGIC=102;
 SELECT  *  FROM  MOVIES, ACTORS
    WHERE  MOVIES.TITLE  =  ACTORS.TITLE;
QUIT;
```

Log Results

```
PROC SQL MAGIC=102;
  SELECT * FROM MOVIES, ACTORS
    WHERE MOVIES.TITLE = ACTORS.TITLE;
NOTE: PROC SQL planner chooses merge join.
QUIT;
NOTE: PROCEDURE SQL used (Total process time):
    real time        0.15 seconds
    cpu time         0.04 seconds
```

Specifying MAGIC=103

The following SQL procedure code and corresponding SAS Log shows the MAGIC=103 option being specified to influence the optimizer in selecting a hash join algorithm for executing the join query.

SQL Code

```
PROC SQL MAGIC=103;
  SELECT * FROM MOVIES, ACTORS
    WHERE MOVIES.TITLE = ACTORS.TITLE;
QUIT;
```

Log Results

```
PROC SQL MAGIC=103;
  SELECT * FROM MOVIES, ACTORS
    WHERE MOVIES.TITLE = ACTORS.TITLE;
NOTE: PROC SQL planner chooses merge join.
NOTE: A merge join has been transformed to a hash join.
QUIT;
NOTE: PROCEDURE SQL used (Total process time):
    real time        0.15 seconds
    cpu time         0.04 seconds
```

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 evaluates the FROM clause references of two or more tables. When it does, a 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 the WHERE-clause 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 22 rows and the second table with 13 rows. The result of these two tables would consist of 286 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 (or internal representation) 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.

MOVIES
Title
Length
Category
Year
Studio
Rating

| ACTORS |
| --- |
| Title |
| Actor_Leading |
| Actor_Supporting |
| |
| |

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 MOVIES, ACTORS;
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 (TITLE) in the following diagram.

MOVIES
☞Title
Length
Category
Year
Studio
Rating

← = →

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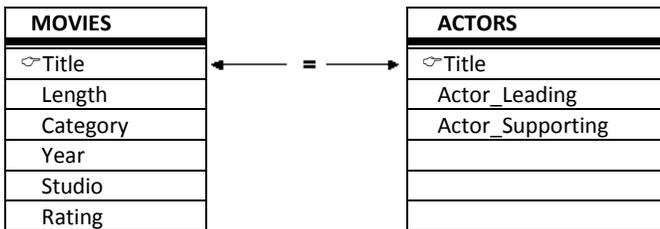
The next SQL code references a join on two tables with TITLE specified as the connecting column. In this example, tables MOVIES and ACTORS are used. Each table has a common column, TITLE which is used to connect rows together from each when the value of TITLE is equal, as specified in the WHERE clause.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

SQL Code

```
PROC SQL;
  SELECT *
    FROM MOVIES, ACTORS
      WHERE MOVIES.TITLE  =  ACTORS.TITLE;
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.

MOVIES		ACTORS
☞Title	⟷ = ⟶	☞Title
Length		Actor_Leading
Rating		Actor_Supporting
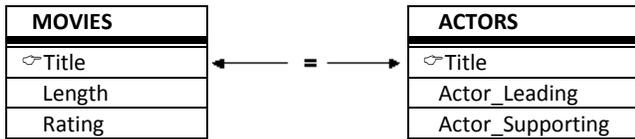
The following SQL code illustrates a join on two tables with TITLE specified as the connecting column. The table aliases are specified in the SELECT statement as qualified names, the FROM clause, and the WHERE clause.

SQL Code

```
PROC SQL;
  SELECT  M.TITLE,
          M.LENGTH,
          M.RATING,
          A.LEADING_ACTOR
    FROM MOVIES M, ACTORS A
      WHERE M.TITLE = A.TITLE;
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 at a time. 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. Essentially the resulting set of data from an outer join process contains rows that "match" the ON-clause plus any "unmatched" rows from either the left, right, or left-and-right tables.

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 cannot 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.title,
            leading_actor,
            rating
     FROM MOVIES
       LEFT JOIN
          ACTORS
      ON movies.title = actors.title;
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.title,
            actor_leading,
            rating
     FROM MOVIES
       RIGHT JOIN
          ACTORS
      ON movies.title = actors.title;
QUIT;
```

Debugging SQL Queries

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, below).

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.

SQL Code

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

```
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. This internal layout representation illustrates the converted PROC SQL code as a pseudo-code. Inspecting the tree output can frequently provide a greater level of understanding of what happens during SQL processing.

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:
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 by using **Hash** techniques. The **BUFFERSIZE=** option can be used to let the SQL procedure take advantage of hash techniques on larger join 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 additional memory is available to conduct the join reducing the number of data swaps the SAS System has to perform from the slower secondary storage.

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 is a wonderful tool for SAS users to explore and use in a variety of application situations. This paper has presented a few of the 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 views; accessing the contents from Dictionary tables; and joining two or more tables to explore data relationships.

References

Lafler, Kirk Paul (2013). *PROC SQL: Beyond the Basics Using SAS, Second Edition*, SAS Institute Inc., Cary, NC, USA.

Lafler, Kirk Paul (2011), *"Powerful and Sometimes Hard-to-find PROC SQL Features,"* PharmaSUG 2011 Conference, Software Intelligence Corporation, Spring Valley, CA, USA.

Lafler, Kirk Paul (2010), *"Exploring Powerful Features in PROC SQL,"* SAS Global Forum (SGF) Conference, Software Intelligence Corporation, Spring Valley, CA, USA.

Lafler, Kirk Paul (2009), *"Exploring DICTIONARY Tables and SASHELP Views,"* South Central SAS Users Group (SCSUG) Conference (November 8th – November 10th, 2009), Software Intelligence Corporation, Spring Valley, CA, USA.

Lafler, Kirk Paul (2009), *"Exploring DICTIONARY Tables and SASHELP Views,"* Western Users of SAS Software (WUSS) Conference (September 1st – September 4th, 2009), Software Intelligence Corporation, Spring Valley, CA, USA.

Lafler, Kirk Paul (2009), *"Exploring DICTIONARY Tables and SASHELP Views,"* PharmaSUG SAS Users Group Conference (May 31st – June 3rd, 2009), Software Intelligence Corporation, Spring Valley, CA, USA.

Lafler, Kirk Paul (2008), *"Kirk's Top Ten Best PROC SQL Tips and Techniques,"* Wisconsin Illinois SAS Users Conference (June 26th, 2008), Software Intelligence Corporation, Spring Valley, CA, USA.

Lafler, Kirk Paul (2008), *"Undocumented and Hard-to-find PROC SQL Features,"* Greater Atlanta SAS Users Group (GASUG) Meeting (June 11th, 2008), Software Intelligence Corporation, Spring Valley, CA, USA.

Lafler, Kirk Paul (2008), *"Undocumented and Hard-to-find PROC SQL Features,"* PharmaSUG SAS Users Group Conference (June 1st - 4th, 2008), Software Intelligence Corporation, Spring Valley, CA, USA.

Lafler, Kirk Paul (2008), *"Undocumented and Hard-to-find PROC SQL Features,"* Michigan SAS Users Group (MSUG) Meeting (May 29th, 2008), Software Intelligence Corporation, Spring Valley, CA, USA.

Lafler, Kirk Paul (2008), *"Undocumented and Hard-to-find PROC SQL Features,"* Vancouver SAS Users Group Meeting (April 23rd, 2008), Software Intelligence Corporation, Spring Valley, CA, USA.

Lafler, Kirk Paul (2008), *"Undocumented and Hard-to-find PROC SQL Features,"* Philadelphia SAS Users Group (PhilaSUG) Meeting (March 13th, 2008), Software Intelligence Corporation, Spring Valley, CA, USA.

Lafler, Kirk Paul (2007), *"Undocumented and Hard-to-find PROC SQL Features,"* Proceedings of the PharmaSUG 2007 Conference, Software Intelligence Corporation, Spring Valley, CA, USA.

Lafler, Kirk Paul and Ben Cochran (2007), *"A Hands-on Tour Inside the World of PROC SQL Features,"* Proceedings of the SAS Global Forum (SGF) 2007 Conference, Software Intelligence Corporation, Spring Valley, CA, and The Bedford Group, USA.

Lafler, Kirk Paul (2006), *"A Hands-on Tour Inside the World of PROC SQL,"* Proceedings of the Thirty-first Annual SAS Users Group International (SUGI) Conference.

Lafler, Kirk Paul (2005), *"A Hands-on Tour of the 5 Most Exciting Features Found in PROC SQL,"* Proceedings of the Thirteenth Annual Western Users of SAS Software Conference.

Lafler, Kirk Paul (2005), "*Manipulating Data with PROC SQL*," Proceedings of the Thirtieth Annual SAS Users Group International (SUGI) Conference.

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.

Acknowledgments

Trademarks

SAS and all other SAS Institute Inc. product or service names are registered trademarks or trademarks of SAS Institute Inc. in the USA and other countries. ® indicates USA registration. Other brand and product names are trademarks of their respective companies.

About the Author

Kirk Paul Lafler is consultant and founder of Software Intelligence Corporation and has been using SAS since 1979. He is a SAS Certified Professional, provider of IT consulting services, trainer to SAS users around the world, and sasCommunity.org emeritus Advisory Board member. As the author of five books including PROC SQL: Beyond the Basics Using SAS, Second Edition (SAS Press 2013), Kirk has written more than five hundred papers and articles, been an Invited speaker and trainer at four hundred-plus SAS International, regional, special-interest user, local and in-house group conferences and meetings, and is the recipient of 23 "Best" contributed paper, hands-on workshop (HOW), and poster awards.

Comments and suggestions can be sent to:

Kirk Paul Lafler
Senior SAS Consultant, Application Developer, Data Analyst, Trainer and Author
Software Intelligence Corporation
E-mail: KirkLafler@cs.com
LinkedIn: http://www.linkedin.com/in/KirkPaulLafler
Twitter: @sasNerd