Structured Query Language: Logic, Structure, and Syntax

Sigurd W. Hermansen, Westat

Abstract

Three basic themes intertwine throughout the Structured Query Language (SQL). SQL statements provide a convenient way to translate the traditional logic of relations among data items to a form closer to machine languages. The structure of a SQL statement adheres to patterns that contain variations on the same patterns. SQL syntax makes the most of a few key terms. This sequel to 'Ten Good Reasons to Learn SAS® SQL' reviews the advantages of more logical, structured, and simpler programming languages, but, unlike its predecessor, recognizes that most beginning programmers who are learning SAS already know the critical elements of SAS SQL. This gentle introduction to SAS SQL helps beginning programmers understand the foundations of a pragmatic, robust, and extensible programming method.

Introduction

You say 'SQL', and I say 'sequel' ... Let's call the whole thing SASequel.

The SAS data step language and the Structured Query Language (SQL) evolved concurrently and relatively independently. By some accounts, IBM dropped PL/1 and developed predecessors to SQL as a programming interface to emerging database management systems. The SAS Institute dropped IBM assembly language and modified PL/1 to support data preparation for statistical procedures on non-IBM platforms. Today SAS SQL is becoming increasingly popular among SAS data step programmers, and the SAS data step language and its extensions are making inroads as a toolset for extracting, analyzing and displaying information stored in different databases.

This introduction to SAS SQL emphasizes SQL and fundamental logic of relations among data items. The SAS System provides a sound basis for learning both. Those new to SQL and SAS will learn to envision data files and records as relations. Those who know SAS will discover that SAS datasets are relations.

Logical Relations among Data Items

Languages bind labels to classes of things known in the abstract as 'entities'. In English-speaking countries, children learn to relate the label 'Dogs' to a special class of animals. The attributes of canines distinguish instances of the general class 'Dogs' from 'Cats', 'Bears', and other classes of carnivores:

```
Dogs ⇐ (legs=4, legsShape='straight', nose='long',
        teeth='fangs', sound='bark', ...)
```

Once children learn to recognize dogs, they learn to relate the names and other attributes of particular instances of the class of dogs. Each child who has an interest in dogs develops a mental list of the names of the dogs he or she knows, and relates each name to attributes that characterize the dog that has that name:

Dogs

name	color	height	length Tail	length Coat
Freckles	mottled	2.5	long	long
Jigs	white	1.5	short	short
Huckle-berry	white	2	long	short
Beauty	black	2.5	short	long
Fletcher	black	2.5	short	long

A child's list of dogs has all of the properties of a 'relation'. A relation expressed as a table has columns representing dimensions or 'attributes' of an entity of class Dogs. Each row of the Dogs table contains attributes of a different instance of Dogs.

Every value in the table relates to Dogs. Each column has a domain of possible values. Each value of an attribute has a relation to each of the other values in the same row. Values in a

column belong to the same set of values or 'domain'.

SQL operates exclusively on data structures with columns and rows. In concept SQL scans table rows one by one and decides whether to keep all, part, or none of the data items in each column. This may sound restrictive, but those who fostered the development of SQL understood that tabular data structures can represent all data, and that element-by-element processing, though tedious, can replicate the results of all programs.

Procedural programming systems put data in a physical sequence, often by sorting on key variables, to prepare a file for efficient processing. In contrast, SQL uses logical orderings of data in one or more columns (also called 'keys') to locate and aggregate data in other columns and in other tables. Primary keys have distinct values in rows and, by virtue of that, identify instances of an entity represented by a table. In Dogs the column labelled 'name' identifies the dogs in the table. In this sense, 'name' serves as a logical key to Dogs. If one knows the name of a dog in the child's mental relation of Dogs, it serves as a key to the other attributes on the same row as that name. Although some columns contain 'natural' keys, any column may at one time or another act as a key.

Data items contained in 'cells' of tables interrelate in a way that children find logical and easy to understand. The data themselves identify each instance of an entity. A column label names an attribute and, to some extent, identifies a type of data. A row contains a set of attribute values that belongs to an instance of an entity. Values appearing in one row of a table belong to one instance of the entity represented by that table. At the point that SQL scans a row in a table, a column name refers to the data item found at the intersection of the column and the row. The 'cell' identified by a column name and a row in a table points to a specific item of data, much as x and y coordinates point to a landmark on a map.

A relational table, characterized by column labels and distinct primary key value(s) in each row, holds data in a structure that makes it easy to ask questions about data items and simple to get correct answers. Using column and row guidelines, we can ask structured questions such as ....

What answer do I get when I .....
Select color from Dogs where name='Beauty' ?

To answer this first question, SQL checks the 'name' column row by row for the value 'Beauty'. That value in this instance of Dogs (one of many possible tables containing data on dogs) appears on the same row as the value 'black' in the color column. In this instance of the Dogs table, the name value identifies each instance of Dogs, so the answer to the question fits into a single cell.

What answer do I get when I .....

```
Select name from Dogs
 where lengthTail='short' ?
```

Will the answer to this second question in this instance of Dogs fit as well into a single cell? SQL checks each row of this instance of Dogs for the value 'short' in the 'lengthTail' column. This yields

name
Jigs
Beauty

a column of only two values, but more than can fit in a single cell. Suppose we had asked,

What answer do I get when I ask.....
Select name, lengthCoat from Dogs where lengthTail='long' ?

That query (to use a quaint and more formal synonym for 'question') yields

name	lengthCoat
Jigs	short
Beauty	long

from the current instance of Dogs. This subset of Dogs will not fit in a single column. It fits into a smaller table than Dogs, but a table nonetheless.

If we ask forms of queries about data in tables, the answers will always fit into tabular data objects: tables, columns, or cells. Queries that yield degenerate tables, columns or cells, have special uses in SQL queries.

This discussion of dogs, relations, and queries has taken us surprisingly deep into SQL programming. Initially we arranged a relation of data items into a table that preserves important relations in (column, row) coordinates. We observed that data in key columns identify instances of an entity. From there, we proceeded to a structured form of query. A query in the standard form

```
Select <column attributes> from <table>
where <condition>
```

yields as an answer at most another table, or, in special cases, a column of data, or a cell containing a single or 'atomic' data item. A 'select' query in SQL has the same basic syntax.

As the scope of instances of the relation Dogs increases, values in the name column repeat and thus do not identify uniquely different instances of dogs. To identity which dogs have which dog tags, the local Dog License Bureau orders dog tags stamped in sequence with distinct numbers.

The Bureau routinely types basic information from license application forms into a spreadsheet. The tagID identifies a row that contains a dog's name, the name and telephone number of the dog's owner, and other values of attributes of the dog and the owner. If anyone reads this 'primary key' value off a dog tag, the Bureau can locate it and find out who owns the dog. For this purpose the tagID substitutes for the occasionally indistinct name of the dog.

Dogs

tagID	name	color	expires	Owner	telephone_ Number
12233	Freckles	mottled	7/22/02	Hermansen	555-1212
23344	Jigs	white	12/18/02	Cotton	555-1234
45566	Huckleberry	white	3/28/03	Beagler	555-1324
34455	Beauty	black	10/1/02	Hermansen	555-1212
21231	Fletcher	black	5/15/03	Cocker	555-1432

The Dogs spreadsheet actually contains a relation between an owner and the owner's telephone number as well as a relation between a Dog and a set of its attributes. This informal design of a database works OK most of the time. As before, the Bureau can find the owner and telephone number related to a tagID and all licensed dogs owned by Hermansen by asking the questions,

What answers do I get when I ...

```
Select owner,telephone_number
from Dogs where tagID=33455 ?
Select tagID,name
 from Dogs where owner='Hermansen'  ?
```

Although the spreadsheet program cannot process questions of this form, the Base product of the SAS System provides PROC SQL for processing SQL queries and tools for converting the Dogs spreadsheet to a table. The SAS SQL engine (compiler) understands the syntax,

```
PROC SQL;
Select owner,telephone_number
 from Dogs where tagID=33455 ;
```

```
Select tagID,name, from Dogs where
owner='Hermansen';
QUIT;
```

SAS reads these statements, opens up the SAS table object (dataset) named Dogs, matches the column names to the references in the SQL query, generates a plan for searching for a solution, and finds and displays the column values in the rows in the solution.

SAS SQL can also process general instructions for creating the shell of a tabular dataset Dogs and for inserting rows into Dogs:

```
Create table Dogs (idTag num(6).,
          name   char(15) format=$15.,
          color    char(8),
          owner  char(15),
  telephone_number  char(15);
```

SAS SQL stores data in SAS® datasets that hold only two types of data: character (char) and numeric (num). The formats tell SAS how to display a column value.

To check that the SQL compiler has understood your declarations, direct the same general form of question to a virtual table containing 'metadata' (data tables containing attributes of data tables). In the SAS System, a table named 'columns' in a library named 'dictionary'contains SAS dataset column-names in the column 'name'. The column 'format' contains values of formats. The column that contain table names has the name 'memname'. (The UPCASE() function changes 'Dogs' to 'DOGS' to match the case of the memname column in dictionary.columns. (Computers take things very literally!)

```
Select name,format from dictionary.columns
where memname=UPCASE('Dogs');
```

The syntax conforms to the general (ANSI) standard for SQL. The format, dictionary, and memname column-names, and the function name belong specifically to the SAS System.

The spreadsheet converted to a SAS dataset and SAS SQL support basic requirements for a dog license database, but soon a progression of new developments exposed the weaknesses of the Dogs spreadsheet. First, the owners of Rex, the Wonder Dog applied for a dog license. Rex, a valuable property, had three owners and each insisted on being listed in the database. One complained, 'What if Rex gets loose and the other owners aren't at the phone number you have for Rex?'

In an effort to appease Rex's owners, the Bureau inserted a row for each of Rex's owners. This quick fix lasted only until someone noticed that the query,

```
Select count(tagID) as count from Dogs
 where expires between 9/1/02 and 9/30/02;
```

exceeded by two the actual number of tags due to expire in September of 2002. A more careful study of the relations among data items showed that more than one dog could belong to the same owner in the original Dogs table, and that would not distort the counts of tags. But when Rex's tagID had to appear in three different rows of Dogs to represent the fact that Rex has three different owners, the relation between dogs and owners became many dogs linked to many owners. The Dogs table no longer represented the relation between dogs and their attributes; it represented the relations between dogs and attributes of dogs, and between owners and attributes of owners, and between dogs and owners. To make matters worse, the owner of Trixie, a Miniature Poodle, insisted on having both her home and work phone numbers listed in

Dogs, so the Bureau could reach her anytime day or night if anything happened to Trixie.

The Bureau made an effort to accommodate these requirements by adding columns for repeats of dogs, owners, and telephone numbers. A modified version of Dogs had columns,

tagID1, name1, color1, expires1, owner11,
telephone_number111,
telephone_number112,....,
owner12, telephone_number121, ...
tagID2, name2, color2, expires2, owner21,
telephone_number211,
telephone_number212,....,
owner22, telephone_number221, ...

Shortly after that the person who maintained Dogs said that she had no idea where to add new tagID's and how to track expires, and she left for another job in the Bureau of Sanitation. While walking out the door, she said that she "... would have less garbage to deal with there."

A more thoughtful redesign of the original Dogs spreadsheet took into account the various relations among dogs and owners. It seemed reasonable to maintain the relation of a dog and its color attribute in the same table.

For all practical purposes a dog's coat has a characteristic color, more than one dog can have the same color of coat, and coat colors don't have dogs as attributes. A so-called 'one-to-many' relation seems typical of the relation between a table key and another attribute in that table. A query may let us observe a one-to-many relation in an instance of Dogs:

```
Select tagID from Dogs
 group by tagID having count(*) >1;
 Select color from Dogs
 group by color having count(*) >1;
```

A SQL 'group by' in the first query forms a group of rows for each distinct value of tagID. The 'having count(*) > 1' clause eliminates all groups of rows containing only one row. The asterisk, '*', denotes all columns selected from the columns in Dogs, in this case tagID only. If tagID acts as a 'primary key', it identifies distinctly each row in Dogs, therefore each group will have a single row, and no group can satisfy the HAVING clause. This means that the first query will not yield a table, nor a column, nor a cell, but an empty result.

In the second query an attribute of Dogs replaces the tagID. Nothing in the expected relation between tagID and color would lead us to expect a one-to-one relation between them. If the second query yields something other than an empty result, tagID has a many-to-one relation to color in Dogs.

The converse does not hold. An empty result of the second query merely shows that in that instance of Dogs, each row has a different value of the attribute color. Any new row of data added to Dogs may introduce a one-to-many relation between tagID and color.

What about other a relation between an identifying key other than tagID, such as owner, and still another attribute of Dogs, such as a telephone number? Some of the pitfalls of allowing this have already come to light. Better to split the spreadsheet into a Dogs table and an Owners table. Again, SAS SQL makes it easy to select different sets of columns ('project columns') into different tables. Use variants of the CREATE clause,

```
Create table Dogs as
select tagID,name,color,expires  from Dogs;
```

```
Create table Owners as
select owner,telephone_number as
telephone_home,'(  )  -  ' as
telephone_work;
```

Projecting the Original spreadsheet into two tables disentangles the complex relations among attributes of Dogs and attributes of Owners. SQL facilitates renaming of columns, definitions of new columns, and, by omission, deletion of columns. The number of owners per dog and the number of dogs per owner does not inhibit the Bureau's efforts to obtain complete lists of dogs and owners. How did the Bureau find a way to represent the complex many-to-many relation between Dogs and Owners? Since creating separate table for attributes related to Dogs and attributes related to Owners, it made sense to create a separate table for the relation between Dogs and Owners. At minimum a new Dogs_Owners table would have both tagID and owner in the Dogs_Owners, and only the two identifiers taken together form a distinct key that can interrelate the Dogs and Owners tables. Each distinct pair of tagID and owner appears on the table. After checking to make sure that the tagID column in Dogs contains distinct values, we *project* the tagID and owner columns into an initial instance of a Dogs_Owners table:

```
Create table Dogs_Owners as
 select tagID,owner from Dogs;
```

This query preserves the relation between dogs and owners that appeared in the original Dogs spreadsheet. A logical data model ties together the new scheme of tables:

Dogs	***Dogs_Owners***	***Owners***
tagID	tagID	owner
	owner	

This bare bones version of the Dog License data model shows only the names of relations (implemented as tables) and the identifying keys. The keys alone interrelate the tables. Other attributes in the tables follow the keys.

The key to Dogs_Owners, tagID + owners remains distinct so long as each row has a different tagID or owner. Now the Bureau can add owners to the Owners table and pair tagID with owner to represent multiple owners of Rex in the Dogs_Owners table. First, the Bureau has to enter Rex's tagID paired with each name of another owner of Rex in separate rows of a spreadsheet and convert it to a SAS dataset (D_O). Then the query

```
Create table Dogs_Owners from
(select * from Dogs_Owners
        union corresponding
 select * from D_O);
```

appends the rows in D_O to the rows in Dogs_Owners. A UNION query combines two tables by appending rows. In this simple case, the rows do duplicate one another when combined, and the columns have the same names and contain the same type of data. The UNION modifier CORRESPONDING tells the SQL compiler to match columns by name.

Note that this instance of a UNION query appends the yields of 'in-line view' (aka 'nested query') SELECT statements. A self-contained, innermost query always executes first. The query that contains it operates on the product of a nested query. (Parentheses around nested queries make intent to nest a query obvious to the SQL compiler.) In the query

```
Create table Owners select * from
 ((select * from Owners)  union corr
  (select distinct owner from D_O));
```

the modifier DISTINCT in the nested query proves redundant. DISTINCT

eliminates duplicate rows, but UNION also eliminates duplicate rows from its yield. The query itself puts the owners in D_O in new rows of Owners. The attributes of Owners related to the new owners stay missing until the Bureau updates them. An UPDATE query

```
Update Owners set
telephone_number='555-1212'  where
owner='Xu';
```

updates a table one row at a time. Alternatively, the Bureau could use an ALTER query

```
Alter  table D_O add telephone_number
char(15);
```

and add telephone numbers to D_O. Then an INSERT query

```
Insert into Owners from (select * from D_O);
```

would put the updated owner rows on Owners.

To select the telephone number for an instance of Owners that links through Dogs_Owners to a tagID requires a JOIN query. Let us start simple by finding all tagID,owner pairs in Dogs_Owners that link to Rex's tagID (112328). The query

```
Select Dogs.name,Dogs_Owners.*
from Dogs inner join Dogs_Owners
on Dogs.tagID=Dogs_Owners.tagID
where Dogs.tagID=112328;
```

first *projects* out all of the columns in Dogs_Owners and only the name column in Dogs. Once again, the asterisk ('*') expands to the complete list of columns in Dog_Owners. SQL then compiles the INNER JOIN into a program that subsets Dogs to the row containing tagID=112328, which as we know has 'Rex' in the name column, and searches each row in Dogs_Owners for a match of tagID to 112328. Upon finding a match, the program writes the name from Dogs ('Rex') and the values in all columns of Dogs_Owners to a tabular output object.

Now, to find all telephone numbers for all instances of dogs in Dogs, the query

```
Select
t1.tagID,t1.name,t2.telephone_number
from ((select * from Dogs) inner join
(select * from Dogs_Owners) on Dogs.tagID
= Dogs_Owners.tagID)) as t1 inner join
(select * from Owners) as t2 on
t1.owner=t2.owner;
```

embeds a query of the same form as the immediately prior JOIN as the value of the argument of a from clause. The nested query links each name of a dog to its owner. With the yield of that in-line view, another simple query can link owner to telephone number(s).

The bureau can also start with attributes related to owner in Owners and work back to identifiers of instances in Dogs. A COMPLEMENT query

```
Select tagID from Dogs_Owners
where owner not in
(select owner from Owners
where input(telephone_number,3.)=555);
```

initially in the nested query yields a list of owners whose telephone exchange is 555. The program compiled by SQL may actually produce an ordered column or owner values or an index. It then scans each row of Dogs_Owners and looks up owner in that row on the list or index. Only if the program fails to locate the owner in the list or index will it produce a new row in the output object with the tagID value in the tagID column.

A relational data model guides navigation from columns in one table to columns in another. SQL contributes both to reshaping data sources into relational tables and to linking values across tables.

At this stage we have seen much of SQL in action. Perhaps more important than that, we have traced through some of the logic behind what has become a standard for database programming. Tabular data objects represent relations among data items, have a fixed number of columns (dimensions), and expand to any number of rows in no set physical order. Rows in relational tables represent instances of an entity. Columns in tables can act as keys that identify each instance. SQL selects and redefines rows, scans rows of a table, and determines which rows to output to a tabular data object. WHERE, ON, and HAVING clauses subset tables and linked views of more than one table. SQL implements the standard relational and Boolean algebras to evaluate expressions that compare data items in a row to each other, to constants, and to linked data items in other tables. In essence, a SQL program subsets and combines tables. JOIN (INTERSECT), UNION, and COMPLEMENT queries within SELECT statements implement basic set operations on sets of database keys. For those still looking for practical applications of 6th grade modern math, look no longer. SQL reduces database programming to that scope and level.

The Structure of a Structured Query

SQL has much in common with Lego®. The blocks that you use to build a table look the same as the blocks that you use to build a link between tables. In a SQL program, if a query produces a tabular object, we can use it as a table reference. In fact, if a query produces a column object, as in the query nested after the IN operator in the COMPLEMENT query, we can use it as a column object. Bordering on the bizarre, we can even substitute a SELECT query for an argument of a SELECT query. All of the SELECT queries that we have seen so far have column lists consisting of column names or constants. The CASE ... WHEN ...clause

```
Select tagID,case when
input(expires,date8.) < today() then
'Overdue' else ' ' end  as status from Dogs;
```

substitutes either “Overdue” or blank into status, depending on the outcome of the condition. We could also produce the same result by writing:

```
Select t1.tagID,
(select 'Overdue' from Dogs as t2
where input(t2.expires,mmddyy8.) < today()
and t1.tagID=t2.tagID)as status
from Dogs as t1;
```

The second version of the query illustrates as well a ‘correlated subquery’. The alias ‘t1' in the subquery cannot be resolved prior to execution of part of the outer query. A self contained subquery would execute prior to the outer query.

The recursive nature of SQL gives it some of the advantages of Unix, Lisp, and Prolog, but makes it almost too easy to continue nesting subqueries until either the SQL compiler has too many datasets open or the programmer gets lost in his or her own maze. SAS SQL makes it convenient to create a VIEW of a self-contained subquery. Say we store a VIEW in the WORK library

```
Create view  555ownerVW as
select owner from Owners
where input(telephone_number,3.)=555;
```

A COMPLEMENT query might reduce to

```
Select tagID from Dogs_Owners
where owner not in 555ownerVW;
```

Many of the examples of queries in the Logic of Relations section went to an extreme to show how queries can substitute for dataset references. The recursive and modular nature of SQL makes it easier to learn the basic components and fairly easy to extend to more complex problems as a programmer gains a deeper understanding of how a SQL compiler processes and improves queries.

SQL Syntax

Each flavor of SQL, including SAS SQL, has definitive rules for constructing valid statements. SQL sticks close to basic ways of specifying the relational operations that inspired its immediate ancestors.

The basic keywords SELECT and FROM produce a subset of a tabular data object based on column definitions that follow SELECT and an incoming tabular object identified by the expression that follows FROM. An optional WHERE clause may eliminate rows from the tabular object defined by the SELECT and FROM clauses. JOIN ..ON .. and UNION keywords act as the fundamental INTERSECT and UNION operators used to combine sets of elements. Since sets have no duplicates of elements, the set operations produce crisp sets when operating on primary keys, but turn fuzzy when operating on indistinct keys.

So-called modifiers change the actions of the SQL keywords. SELECT DISTINCT ... eliminates duplicate rows (not necessarily duplicate keys) in the tabular object produced by the SELECT statement. The LEFT [RIGHT] and FULL versions of JOIN queries modify the INNER JOIN default. For tables L and R the LEFT JOIN yields (L INNER JOIN R ON key) UNION CORR (L COMPLEMENT R), while the FULL JOIN yields (L INNER JOIN R ON key) UNION CORR (L COMPLEMENT R) UNION CORR (R COMPLEMENT L). SQL syntax simplifies declarations of these composite relations, but not the difficulty that beginning SQL programmers have in understanding what the LEFT and FULL JOIN's do.

Beginning programmers need intensive practice in subsetting and recombining tabular data objects. Much of database programming, independent of platform and programming environment, involves SELECT clauses and chains of JOINS of subsets. As in the simple model of the dog license database presented earlier, tagID in Dogs links to tagID in Dogs_Owners, and the related owner in Dogs_Owner links to the owner in Owners. This obvious but powerful 'transitive logic' unifies the schema of relational databases. At first it pays to focus on the essential syntax outlined here. ANSI SQL includes a much richer set of rules, but all depends on how well the programmer understands the basic table subsetting and combining operations.

For the experienced SAS data step and procedure programmer, SAS SQL blends well with data steps and procedures. The SELECT clause and JOINS empower the SAS programmer to reach into external sources of data, reshape data files, and improve the precision and adaptability of SAS programs. With a few exceptions, SAS functions, formats, informats, and operators carry over verbatim. It does not take long to catch on to useful SQL features, such as qualifying column names with table names, type converting and renaming column variables, or using functions, formats, and informats in the ON conditions that replace BY groups. Accomplished SAS programmers have little new to learn and much to gain. Several recent SUGI and

regional conference papers have stepped through the many similarities and few differences between data step and SQL syntax. With few exceptions, SAS procedures and data access engines have only inferior substitutes in SQL. A SAS datastep programmer who becomes proficient in SQL will have no trouble writing basic SQL queries and passing them through to RDBMS servers.

We hope that placing the SQL syntax topic at the end of this introduction to SQL, rather than at the beginning, now makes sense. Once a programmer has an understanding of the logic of relations and a feel for the structure of a SQL query, SQL syntax falls in place nicely. A SQL programmer can find many different ways of writing a SELECT statement that conforms to the rules of the SQL language. In a good SQL solution, statements of basically the same form connect and stack in a logical structure.

Conclusion

A SQL program restates pre-existing logical relations among data items as tabular data objects, and then specifies how to subset and combine tabular data objects to produce a result. Basic relational algebra, set logic, and Boolean logic have important roles in SQL programs. The structure of a SQL query encourages reuse of the same basic structures to construct a new logical structure that includes a solution in the form of a tabular data object. It takes some practice to get used to the syntax of SQL, but beginning programmers can focus initially on basic variations on the SELECT statement and the FROM, WHERE, JOIN ...ON, and GROUP BY ... HAVING clauses. Experienced SAS programmers will find powerful new features, but may focus on the SQL queries that complement SAS data steps and procedures. By design the data step and SAS PROC SQL queries share SAS datasets and almost the same function, format, and informat libraries. As SAS SQL continues to gain in popularity, it will become a required tool for good SAS programming.

Disclaimer

The contents of this paper are the work of the author and do not necessarily represent the opinions, recommendations, or practices of Westat.

Acknowledgments

Many contributors to SAS-L have through no fault of their own contributed to this effort. My colleagues at Westat deserve special thanks.

References

Hermansen, S. “Ten Good Reasons to Learn SAS® Software's PROC SQL", Proceedings SUGI 1997, San Diego, 1997.

Whitlock,I. "PROC SQL - Is it a Required Tool for Good SAS®" Programming?" Proceedings SUGI 2K, Paper 60-26

Author Contact Information

Sigurd W. Hermansen
WESTAT, An Employee–Owned
Research Corporation
1650 Research Blvd.
Rockville, MD 20850
USA
phone: 301.251.4268
e-mail: hermans1@westat.com