TEN GOOD REASONS TO LEARN SAS® SOFTWARE'S SQL PROCEDURE

SIGURD W. HERMANSEN, WESTAT,ROCKVILLE,MD

ABSTRACT

SAS PROC SQL works interchangeably with and complements data step programs and other procedures, but does it offer any new capabilities? Some say no. This tutorial presents a series of examples of PROC SQL solutions to problems that would prove very difficult or cumbersome to implement without SQL. Examples introduce the use of table and column aliases; catalogued views; inline views; host-variable references; summary functions applied by group; and complex joins.

Practical applications of SQL constructs make a convincing case for learning these useful tools. A few examples of some data step constructs that prove difficult to match in SQL round out the discussion.

During the few years since the introduction of SAS PROC SQL, the use of SQL for database access has seeped slowly into the large base of SAS programmers. Slowly, from my point of view, because I believe that PROC SQL represents a major step forward for the SAS System and that it deserves much wider use.

Several factors have limited wider use of PROC SQL[1]. Beginning SAS programmers tend to learn traditional SAS programming methods first. It takes time to change training curricula, and the standard SAS Language[2] manual introduces methods that date back to the era of procedural languages. Experienced SAS programmers lack the incentive to change. Many have heard on good authority that anything you can do in SQL, you can do in the older SAS language constructs (called "datasteps" here). SQL, the story goes, does not really offer anything new, and it looks strange to boot.

During the last few years I have developed a repertoire of examples showing why beginning and experienced programmers should learn SAS SQL. My colleagues at work have helped me refine the reasons, often with subtle counter-arguments ("That's not right!" and "Your SQL example brought our system down!" and "Is it supposed to print the right answer?") Over time they have persuaded me to distill hundreds of half-brewed arguments into ten good reasons.

For those that have invested a considerable amount of time studying the SAS Language manual, I hasten to add that PROC SQL does not make the SAS datastep obsolete. Following my glossy review of SAS SQL, you will find several good reasons for keeping the old SAS language manual close at hand.

Ten Good Reasons for Learning SAS SQL

REASON #1. *SQL has less pointless stuff for programmers and other users to remember*.

It often takes too long to organize a SAS datastep program. Sometimes the order of statements and options matters and sometimes it doesn't. Tracing through all of the statements and options that define a variable can prove tedious, and forgetting to put a semi-colon at the end of a statement ......

The "S" in SQL stands for "structured". SQL provides a basic structure of keywords and lists. The programmer uses this structure to declare the type and name of the result, the column (variable) definitions, the source data table(s), and the condition for selecting rows (observations). As a result, SQL programs require

- *fewer statements and thus fewer pesky semicolons and other syntax elements (and, of course, no line structure).* A few basic structures work across the board. For instance, the "**C**older **T**han **A** **S**an **F**rancisco **W**inter" clause structure takes care of most of the basic table access, restructuring, and merge operations in Base SAS datasteps:

SAS SQL	*SAS Datastep*
CREATE **T**ABLE ...**A**S	DATA ....
SELECT	(KEEP=...);
FROM	SET/MERGE ....;
WHERE	BY...;
;	IF (subsetting)....;

SQL requires a semicolon only at the end of the clause structure. If you add to the SELECT clause and wrap around to another line, you don't have to worry about moving the semicolon down to the second line. You can write different expressions in a WHERE clause and copy and edit them without losing a semicolon or adding an extra one. More than just a sequence of statements that the programmer strings together to specify a process, a SQL program has a definite purpose and a structure to fit that purpose.

- *column definitions for a table in one place*. The SELECT list not only tells the compiler which of the existing column variables to include in the result table, it also specifies any new columns (including constant expressions); the naming and ordering of the columns; and formats and labels for the columns. All of this appears in one place in the intended order. Commas separate the elements of the SELECT list. An AS keyword makes it easy to specify and recognize a new column name following an existing column name or an expression containing existing column name(s) and/or constant(s). Format and label keywords in SAS SQL help keep the definition of a column variable in one place. For instance, ... SELECT ID,round(price*quantity,.01) AS cost FORMAT=9.2 LABEL="buyer pays" .... puts a complete definition of columns in a SELECT list.

- *fewer confusing options.* In a SAS datastep, for example, the keyword KEEP may appear as a SET statement option, a DATA statement option, or in a KEEP statement. These optional ways to KEEP column variables differ in subtle ways. It takes a real SAS expert to know when to use one method rather than another. These types of options usually arise when developers of compiler programs add new, more general methods, but need to maintain compatibility with procedural options inherited from earlier versions of the language. Designed from the start as a declarative language, SQL has far fewer vestiges of procedural language ancestors.

In SQL a programmer finds the best qualities of SAS data steps plus a simplified structure.

REASON #2. In SQL, a table is a table is a table ... is a view.. is a query ... is a SAS dataset ... is a relation.

When working with one dataset or combining data linked by common keys, SAS datasteps work fine. When working with a complex database, with interrelated tables, SAS dataset programming becomes much more difficult.

In string processing functions in C, everything is a character; in a Unix script, everything is a file. In SQL, everything is a table. We may represent a table as a view, or a query on an external data source, or a SAS dataset. We may think of it as an abstract relation. So long as it resolves to columns of data in rows embodied in a fixed-format data stream and described in a recognized catalog, SQL will accept it as a table.

The role on the table data type in SQL extends to the elements of the lists. The user can, for example, embed another SQL query in a FROM or WHERE clause, as in

```
        SELECT *
        FROM            (SELECT distinct t1.person,t2.outcome
                        FROM sample AS t1,test AS t2
                        WHERE t1.person=t2.subject
                        )
        WHERE outcome in
                        (SELECT distinct diagnose
                        FROM special
                        WHERE agent="VIRUS"
                        )
        ;
```

These so-called "in-line views" add immensely to the expressive power of SQL. Whether in a SELECT, FROM, or WHERE clause, an in-line view can substitute for any argument that represents a table. SAS SQL makes it much easier to combine on different dimensions (say, person, test, outcome) information from different sources.

If embedded in-line views make the SQL program more difficult to construct and understand, then the programmer can define the same views separately and make it look more like a procedural program, as in

```
        CREATE VIEW samplvw AS
        SELECT distinct t1.person,t2.outcome
        FROM sample AS t1,test AS t2
        WHERE t1.person=t2.subject
        ;
        CREATE VIEW speclvw AS
        SELECT distinct diagnose
        FROM special
        WHERE agent="VIRUS"
        ;
        SELECT *
        FROM samplvw
        WHERE outcome in speclvw
```

The programmer can also interchange references to SQL tables, SQL catalog views, SAS work datasets or views already created outside PROC SQL, queries on external data sources, and permanent SAS datasets on different SAS libraries (including EXPORT libraries). Moreover, SAS datastep programs recognize SAS SQL tables and views as equivalent to SAS datasets. The SAS Version 6+ System treats PROC SQL queries interspersed among datasteps and other SAS PROC's the same way it treats PROC MEANS or PROC CATMOD. It interchanges SAS datasets or views seamlessly in datasteps, SQL, and procedures.

Using views rather than temporary datasets to store the results of preprocessing source data has a secondary benefit. It takes almost no time, disk space or other computer resources to create a view. Syntax errors turn up much quicker, and programmer and compiler have a better opportunity to optimize (actually, improve) the efficiency of the query.

Reason #3. SQL's smarter than the average compiler program.

A SAS datastep program may work perfectly until something changes the order of observations in a data source. A MERGE BY operation may run for a time and then fail with a run-time error.

Even more so than other supposedly fourth generation languages (4GL), including SAS datasteps, SQL in SAS or otherwise does not require the programmer to specify details that the compiler program can figure out for itself. For instance, database access languages require users to sort physical files or create indexes. SQL handles these tasks as needed.

Let's start with a short but rich example of a query program that leaves it to the SQL compiler figure out how to order or index data sources. Say you need total sales greater than $1000 per buyer with the highest sales totals listed first. You have a file of purchase orders by itemcode. You have a price list by product ID. The itemcode appears as a substring in the product ID.

This query has to

- match up two data tables on identical values of a substring of product ID in one table and itemcode in another;
- collect together all sales volume values related to the same buyer ID;
- sort the table created by this query in a descending collating sequence so that the highest sales volume appears at the top.

A SAS program based on PROC SORT and datasteps would have to sort the purchase order and price list tables by the key itemcode and product ID substring prior to merging the two and calculating sales volume, sort the result of the MERGE by buyer ID and use PROC MEANS or SUMMARY (with the NWAY option) to add up the sales per buyer, and sort the results one last time by sales volume.

The SQL alternative simply uses the table and column names on the SAS catalog to describe the table required:

```
PROC SQL;
        CREATE TABLE sales AS
        SELECT DISTINCT t2.buyerID,
              SUM(t1.price*t2.quantity) AS salesvol
        FROM pricelst AS t1,qntordr AS t2
        WHERE substr(t1.product,4,7)=t2.itemcode
        GROUP BY t2.buyerID
        HAVING CALCULATED salesvol GE 1000
        ORDER BY CALCULATED salesvol DESCENDING
        ;
QUIT;
```

The SAS SQL compiler

- determines which of several methods should work most efficiently and uses that method to join (equijoin or merge) the two tables;
- determines how to collect the calculated values of salesvol for the same buyerID and add them;
- sorts the sales table by salesvol sum.

All this happens automatically during the compiling and processing of the query.

Many prospective users of SAS SQL have heard that SQL programmers have to know a lot about creating and managing indexes. Not true! Knowing how to create and manage indexes may help make SQL programs more efficient; nonetheless, leaving indexing and other data ordering operations entirely in the "good hands" of the SAS SQL compiler will not do any irreparable harm and may actually lead to more efficient processing.

The developers of SQL began with the premise that requiring the programmer to know anything initially about the order of rows in a table would lead to a lot of unnecessary complications. In some special cases it makes programming more difficult if we can't assume, for example, that after testing for a match on person ID the next record in sequence represents a later record for that person. The validity of a lot of legacy programs depends on this class of assumptions. It turns out, though, that in most cases we can write equivalent programs that require no assumptions about the order of rows in source tables. In doing so we allow programming language developers some room for making compiler programs smarter. Smarter compiler programs such as SAS SQL free us from the details of processes such as sorting, indexing, and verifying that the order of the data matches the process. Having to pay less attention to the details of basic processes should shift our focus to the way in which the program describes the results we want to produce from our data sources.

Moving from partially procedural programming languages to SQL requires this shift in focus. We call SQL a declarative language because the SQL programmer uses it to declare a result and the compiler program constructs a step-by-step process that implements it. All of us do so when we declare a desired product as x=a*b; , rather than telling the compiler how to find the product using an iterative addition procedure that we learned in our first programming course:

```
        x=0; c=1;      do while c <=a; x=x+b; c=c+1;
                       end;
```

We would no doubt agree that a programmer should declare the product as the result of an expression of two numbers. Shouldn't a smart programmer also learn how to declare other classes of results and let the compiler program call the procedures needed to produce it?

REASON #4. *SQL has a richer and more precise collection of methods for combining data*.

Many programming tasks boil down to nothing more than subsetting sets of data in a way that one could describe using simple Venn diagrams. Many SAS programmers find it difficult to visualize the solution and then translate it into a datastep program.

The SQL language makes it easy to express and combine operations on tables. A few basic set operations have equivalent expressions in SQL. These serve as the building blocks:

SET	*SQL EXPRESSION*	*SYMBOL*
intersection	SELECT * FROM t1,t2 WHERE $t1.c_i=t2.c_j$.	$\otimes$
union	SELECT * FROM (SELECT * FROM t1 OUTER UNION CORR SELECT * FROM t2)	$\cup$
complement	SELECT * FROM t1 WHERE $t1.c_i$ NOT IN (SELECT $t2.c_j$ FROM t2)	—

The **intersection** (produced by an equijoin and not exactly the same results as that produced by the SAS SQL set operator, INTERSECT) contains, if not empty, some or all columns from one table and some or all columns from another. It includes only those rows in which key columns in one table match key columns in the other. The key may consist of a single column, such as an ID, or an expression based on columns. If we want to find all matching pairs of patient records and visit records and we have a patient ID in each patient record and a corresponding type of patient ID in each visit record, the intersection defines a new table that contains the patient ID, all of the columns in the patient record and all of the columns in the visit record.

The **union** contains the rows from one table plus those from another. The CORR or CORRESPONDING keyword tells the SQL compiler to match columns in one table to those in the other by column name, not position in the table. To keep the order of columns in a table from becoming a programming issue, it makes sense to use the CORR keyword with UNION as a matter of routine. If we have collected tables of the same form of visit record from two clinics and want to combine them into one file, we could define the union operation.

The **complement** contains all of the rows in one table that do not match on a key at least one of the rows in another table. In a sense, subtracting the keys in one table from those in another leaves the complement. If we have finished checking a subset of all visit records and now want to select the ones that we have not checked from the full set, the complement defines a new subset that excludes the ones already checked.

Putting a SELECT clause, in front of a FROM clause in SQL lets us specify the columns required. Putting a WHERE clause after the FROM clause (or adding conditions to the WHERE $t1.c_i=t2.c_j$ clause in an equijoin) will limit the selection of rows to those that meet the WHERE conditions.

We can use basic set operations to compose other sets. Composite set operations give us the ability to specify the different ways that we might want to combine data tables. SQL includes several concise ways to declare certain composite operations. For example, we may want

1) to match all patient records to visit records, but also include the patient record with visit missing when a patient has no visit records;
2) to match visit records to patient records but also show the visit records that do not match the patient records;
3) to select all of the matching patient and visit records, but also select all patients without visit records and all visits without patient records.

The three composite operations[3] shown below correspond directly to 1)., 2)., and 3). above:

COMPOSITE SET OPERATION	SQL EXPRESSION
1). $(t2 \otimes t1) \cup (t1 - t2)$	SELECT * FROM t1 LEFT JOIN t2 ON $t1.c_i=t2.c_j$
2). $(t1 \otimes t2) \cup (t2 - t1)$	SELECT * FROM t1 RIGHT JOIN t2 ON $t1.c_i=t2.c_j$
3). $(t2 \otimes t1) \cup (t1 - t2) \cup (t2 - t1)$	SELECT * FROM t1 FULL JOIN t2 ON $t1.c_i=t2.c_j$ ..

The union operation also has simple and special forms. The statement

SELECT * FROM t1 UNION CORRESPONDING SELECT * FROM t2

selects the columns that have corresponding names from t1 and t2 and appends the unique rows from both tables. Putting the keyword OUTER just before UNION CORR works much the same, except it also selects columns from either table that do not have matching names in the other table (setting the undefined parts of columns to missing values). OUTER UNION CORR works much like a SET t1 t2 statement in a SAS datastep. Adding the keyword ALL after UNION CORR keeps duplicate rows in the union of the two tables.

SQL has a firm mathematical base supporting its methods of interrelating data tables. The language allows the user to make distinctions among the different methods of joining column values on rows and appending rows from two or more tables.

REASON #5. SAS SQL has a foolproof (almost) method for defining scalar constants during program runtime.

Say you want to compute a single number that represents the average number of days it took to fill an order last year. You can compute this number from a file of order numbers, order dates, and delivery dates. You then plan to subtract this number from the actual number of days it has taken to fill an order this year and sum the differences.

In SAS SQL you can SELECT the average directly into a SAS macrovariable and subtract that macrovariable value from the days to delivery calculated for each order this year. These short SQL queries demonstrate the method:

```
SELECT AVG(DelDate-OrdDate) INTO :AvgDif95 FROM temp95;
< more SQL>
TITLE "95 Avg. Wait: &AvgDif95 ";
SELECT AVG((DelDate-OrdDate) - &AvgDif95) AS DifAvg
LABEL="Diff 96/95 avg"
FROM temp96;
<Other queries with value of  AvgDif95 inserted.>
QUIT;
```

Why "almost foolproof"? You can't use the macrovariable AvgDif95 until after the select statement that defines it finishes running.

REASON #6. *SQL keeps track of which column variables come from which table or view.*

Many questions about SAS syntax and methods concern the difficulties involved in naming, managing, and renaming variables. Many of these difficulties disappear when a programmer has the option to qualify the name of a column variable by a reference to its source table.

SQL makes referring to columns particularly easy by letting programmers assign aliases to table references. (I use the AS after a table name to assign the aliases t1, t2, etc.; I then distinguish column references by qualifying them with the table alias, as t1.x,t2.x, etc.) It doesn't hurt to use qualified column names even when the compiler doesn't really need them, so, as long as we have the alias defined, we do not need to remove them from segments of SQL programs copied from other programs.

Qualified column names become particularly useful when we need to compare values on one row of a table with values on another row of the same table. Say we are looking at a set of test results on samples and we are trying to find different tests that have different results for the same sample. The program

```
SELECT t1.sample,t1.date,t1.seq,t1.result,t2.date,t2.seq,t2.result
FROM samples AS t1,samples AS t2
WHERE t1.sample=t2.sample
   AND (t1.date < t2.date OR t1.seq <t2.seq)
;
```

uses different aliases, t1 and t2, for two references to the same table. SQL treats two references as different views and has no problem with inter-row comparisons. The inequality conditions on the comparison of the date and sequence reference prevents matching on the same row. All this depends on having a method for distinguishing one reference to a column from another by its source.

REASON #7 SQL's easier to write, understand, and modify.

Following a trail of variable names through several SAS MERGE steps involving renaming can prove confusing and lead to errors.

Most users of C/C++, earlier procedural languages, APL, GUI OOP's, or other write-only languages concede that SAS datastep programs have the advantage in ease of writing, understanding and modification. SAS SQL rates even better than SAS datasteps.

Let's look first at continuing data from three datasets using SAS datasteps vs. a SQL query:

```
* Assuming dbx and dby sorted or indexed by ID1;
DATA d1 (KEEP=ID1 test        RENAME=(ID1=ID));
        MERGE   lib.dbx (KEEP=ID1 site
        IN=inx)
                lib.dby (KEEP=subject
                test IN=iny
                RENAME=(subject=ID1));
        BY ID1;
        IF inx AND iny AND site ne "x";
RUN;

PROC SORT DATA=d1;
        BY ID test;

RUN;

* Assuming dbz sorted or indexed by assay;
DATA d2DS (KEEP=ID test outcome);
        MERGE   d1  (IN=in1)
                lib.dbz (IN=inz
                RENAME=(assay=test));
                BY ID test;
                IF in1 AND inz;

RUN;

* Equivalent SAS SQL program;
PROC SQL;
        CREATE TABLE d2SQL AS
        SELECT DISTINCT t3.ID, t3.test, t4.outcome
        FROM (SELECT DISTINCT t1.ID1
                        AS ID,t2.test
                FROM lib.dbx AS t1,lib.dby AS t2
                WHERE  t1.ID1=t2.subject
                        AND t1.site NE "x"
                ) AS t3,
                lib.dbz AS t4
        WHERE t3.ID=t4.ID
         AND t3.test=t4.assay ;
QUIT;
```

The SQL version clearly raises the unit of data to the table level. This basic structure CREATE's a TABLE by SELECT'ing rows FROM a table WHERE each row meets conditions. The SQL programmer declares the characteristics of a data table and the name of the source table. The SQL engine combines the SQL program and data stored in a catalog (the so-called *metadata* in tables that relate the symbolic names of tables and table columns to the locations of data stored in the host system). It produces the table d2SQL as declared in SQL for implementation by the SAS SQL compiler and SAS System.

The SAS SQL query reduces a process description to a state description. The programmer does not have to worry about the order of operations (for example, whether or not the RENAME= precedes the KEEP= in a DATA statement option list). One can assign aliases to columns to avoid ambiguities; by default the columns in views or tables inherit their names from their source tables. Queries take data from one state to another. The SQL programmer declares a new state of the data; the compiler figures out the process required to transform the data to the new state.

REASON #8 *It's easier to write a program to write SQL.*

System developers are increasingly using automated methods to write programs. SAS datastep programs can, for example, read external files and datasets and use the information they contain to write a query on another database or a continuation of the same program.

The standard SQL syntax has a relatively compact and simple description (for example, when expressed in the BNF notation used to describe other ANSI programming language standards). SAS SQL extends the standard syntax primarily by implementing the rich set of functions, informats, and formats of the SAS datastep language.

The structure of a SQL statement consists of keywords followed by comma-delimited lists. This makes it easier to insert table and column names from macrovariables or files.

The ability to use an automated process to write programs is becoming an important feature of programming systems. SAS SQL has a syntax that system developers find simpler and more familiar than SAS datastep syntax.

REASON #9. ***Join SQL and see the rest of the world.***

An object can communicate with another object by sending it a message in the form of a program written in its language or in a common subset of the languages understood by the two objects. Outside of the SAS world, no one understands the SAS datastep language but many understand a subset of SAS SQL.

Whether or not SQL works best in the SAS system, nothing else in the SAS language works at all in a query being sent to another database system. Many other database systems understand a standard subset of SQL. SQL queries have become the standard method for specifying data requests from one database system to another

Client access to SAS server datasets has a better chance of succeeding when the client writes the data request in SQL. The Open Data Base Connectivity (ODBC) method for exchanging data among databases, for example, accepts SAS SQL queries on SAS datasets, but does not accept datastep programs or other procedures.

REASON #10. SQL sounds a lot better than OOPS!

What you call a programming method won't make it more productive, but it could make it easier to defend the budget for your system development project.

SAS SQL fits in between the old mainframe methods and the brave new world of WindowsXX. SQL has the right image for the time.

And isn't image everything? If not so, OOPS would have already evolved into Pragmatic Object-Oriented Programming Systems.

Good reasons to keep your SAS Language manual.

A few reasons seem particularly obvious and important:

- Partitioning a dataset into several subsets using SELECT or IF conditions has a simpler and more direct implementation in a SAS datastep than in SAS SQL;

- The INFILE ... INPUT and FILE ... PUT statements in SAS datastep programs can handle a wide variety of file stream parsing and other tasks related to acquiring data from external files. The standard RDBMS offers many features related to on-line data entry and updating but few tools for capturing data from system files. Entry of data from files generally falls outside the scope of SQL;

- In some applications, implicit ordering of data in a physical file really matters. For instance, a source data file may have a primary key ID, but that ID may contain meaningful data, such as the first three characters of a person's last name and SSN. Sorting the file by a random number and creating a primary ID from the automatic SAS variable _n_ would create both an encrypted ID and a mapping of the encrypted ID to the actual ID. All this depends crucially on maintaining the original ordering of the source file until the procedure finishes creating both the encrypted main file and the mapping files;

- The SET t1 t2 .... method of appending datasets seems simpler than the UNION set operator for those who already know and understand it;

- The SAS datastep has much better tools for creating test data. After declaring a SAS SQL query, a wise programmer writes SAS datasteps to create a test database and tests the SQL program.

SAS SQL, we should add, does not replace the other SAS procedures (FORMAT, COMPARE, PRINT, TRANSPOSE, etc.; SAS/STAT PROC's; IML, MACRO, or other specialized tools. These distinguish the SAS System from RDBMS and OOPS. Fortunately, the SAS System makes it easy to use different methods in harmony. Casting SQL as a PROC has allowed programmers to build SQL gradually into SAS programs. At first, one can try a query that directly replaces an equivalent SAS datastep. In time one might find, as I have, that fewer and fewer programs require SAS as we knew it in the pre-SQL era.

Author's e-mail address: hermans1@westat.com

Acknowledgments

Ian Whitlock, Nancy Odaka, Steven Schweinfurth, Willard Graves, James Ingraham, Duke Owen, and Drew Hayes contributed comments and suggestions that led to substantial improvements in the contents and presentation. Darlene Faiola and Betty Ovington improved the design and layout of text and examples. The author alone takes responsibility for remaining defects. The views presented do not necessarily represent those of Westat, Inc.

References

[1] SAS Institute Inc. (1989) **SAS Guide to the SQL Procedure**, First Edition, Cary, NC: SAS Institute Inc.

[2] SAS Institute Inc. (1990) **SAS Language Reference, Version 6**, First Edition, Cary, NC: SAS Institute Inc.

[3] Some dialects of SQL do not support the full (outer) joins. See Gruber, Martin. (1990) **Understanding SQL** , Alameda, CA: Sybex, pp. 177-181, for SQL programs that compose set operations to achieve the same effect. Also, Appendix D, pp. 400-417, contains a concise BNF description of standard SQL syntax.