

SQL: A Sequel to the Data Step

Heidi Markovitz, Simply Systems, Key Biscayne, FL

Abstract:

This tutorial introduces the SAS implementation of the SQL language. It explains the basic structures and functions used in PROC SQL. Attendees will be shown how to use SQL to extract data from SAS data sets, report on that data, and build new tables and views from extracted data. For listeners in a dating mode, plenty of attention will be given the SELECT statement and the various forms of UNION (or JOIN). Examples will compare the SQL way of data processing with other SAS techniques, including the DATA step and PROC SORT.

What is SQL?

Structured Query Language, commonly called SQL, is a non-procedural language for maintaining data in a Relational Data Base. "Non-procedural" means that the programmer specifies the input and output of the process without describing the sequence used to obtain that output.

SQL was first publicly described in a paper by E. F. Codd in 1976, for use with a prototype Relational Data Base System (RDBMS). Its first commercial implementation was in Oracle® in 1979. Since 1986 SQL has been an ANSI standard language for RDBMS's. Since that time it has also been adopted by the International Standards Institute (ISO) and made a Federal Information Processing Standard (FIPS). Currently some version of SQL is a major component of Oracle®, IBM DB2®, Informix®, and Foxpro® to name a few. Since the beginning of Release 6, it has also been a SAS® PROC.

Why Use SQL?

SQL is a tool in so many packages that the skills you build with it are transferrable. You can use them for new applications in your firm or for helping you get a new job. Conversely, you may have staff or clients who want to use SAS data but are unfamiliar with SAS. If they already know SQL from another package they can quickly begin manipulating SAS data sets using their SQL knowledge. You can even use SAS PROC SQL to process data in competing data base packages, if you license SAS ACCESS®.

Further, sometimes the SQL way of doing things is faster or easier than any other SAS way. This paper will show several examples of performing the same task with and without SQL. You can decide for yourself where SQL is most applicable.

PROC SQL is included in BASE SAS (at no additional charge). Also included in recent releases is a point-and-click (that's Tab-and-Enter for 3270 users) Query Window that guides you in assembling SQL routines. If you can't

figure out SQL syntax, you can use the Query Window to generate actual SQL code that you can edit yourself. The balance of this paper will cover syntax and functions of the SQL procedure and leave the Query Window to other presenters.

Terms Used

SQL uses different terms than original SAS. Here are some definitions and their SAS equivalents, where relevant.

SQL Term	SAS Term	Other Definition
Table	Data Set	File
Row	Observation	Record
Column	Field	Individual item of data
View	N/A	Virtual Table. A way of looking at a combination of parts of one or more tables.
Data Base	N/A	Collection of tables and Views
Relational Data Base	N/A	Data Base in which data from any combination of tables can be linked by any relationship you can imagine.

Sample Data

The sample data in this paper are based on an Inventory Tracking system I originally built for a cosmetics manufacturer using Informix-4GL®. The data sets used here are defined as follows:

rm lot -Numbered lots of Raw Materials used to manufacture batches of cosmetic.

Variables	Type	Length	Format	Label
LOTNUM	Char	10		Lot Number
QTYNOW	Num	8	10.2	Quantity on Hand
QTYRECD	Num	8	10.2	Quantity Received
RMNUM	Char	10		Raw Material #
UNITMEAS	Char	3		Unit of Measure
UNITPRIC	Num	8	7.2	Unit Price

>>> Index MAIN is UNIQUE: RMNUM LOTNUM

batchhead -Batches of each cosmetic manufactured

Variables	Type	Length	Format	Label
BTNUM	Char	10		Batch Number
CUSTID	Char	10		Customer ID
NUMMAT	Num	8	3.	Number of Materials Used
PRODNUM	Char	10		Product Number
QTYREQST	Num	8	10.2	Quantity Requested
UNITMEAS	Char	3		Unit of Measure

>>> Index is UNIQUE: BTNUM

batchdet -Detailed list of each ingredient used in each Batch of cosmetic. For every **batchhead** record, there are one or more **batchdet** records, describing the quantities used and the Raw Material Lots from which each ingredient was taken.

Variables	Type	Length	Format	Label
BTNUM	Char	10		Batch Number
LOTNUM	Char	10		Lot Number
MATSEQ	Num	8	3.	Material Sequence
QTYREQR	Num	8	10.4	Quantity Required
QTYUSED	Num	8	10.4	Quantity Used
RMNUM	Char	10		Raw Material #
UNITMEAS	Char	3		Unit of Measure

>>> Index MAIN is UNIQUE: BTNUM MATSEQ

>>> Index SECOND is NOT UNIQUE: RMNUM LOTNUM

All sample sets are indexed as indicated. This means they can be treated as though they are sorted in the order of the index.

First, A List

The simplest SQL functions involve listing data from tables. Figure 1 shows how to produce a list of all Batch Header records, introducing the most common SQL construct, the **SELECT** clause. Please note that **PROC SQL** always begins with **PROC SQL** and ends with **QUIT**. In between you can insert any number of action statements.

```
SQL Code-
TITLE1 'PROC SQL: List BATCHEAD With Labels';
PROC SQL;
  SELECT *, COUNT(*)
  FROM sesug.batchhead;
QUIT;
```

Output-

PROC SQL: List BATCHEAD With Labels

Batch Number	Product Number	Customer ID	Quantity Requested	Unit of Measure	# of Materials Used
B10000	P0512	HERO	350.00	LB	5
B10010	P0430	HERO	100.00	LB	3
B10020	P0430	DUKES	150.00	LB	3
B10040	P1022	EYESON	200.00	LB	2

Fig.1 SQL Code, with Output - Listing a Table with Labelled Columns

SELECT describes which data you want. In this case, the key symbol ***** indicates that we want all data available from the source, and the key phrase **COUNT(*)** to request a count of the observations processed. The **FROM** clause, which is also vital, describes the source of our data, in this case the **batchhead** table in library **sesug**. As part of SAS, **PROC SQL** uses the standard SAS syntax **libref.data set** in referring to tables. Similarly, the SAS **TITLE** statement is effective on SQL output. Non-SAS implementations of SQL may use different syntax.

```
SAS Code-
TITLE1 'PROC PRINT: List BATCHEAD With Labels';
PROC PRINT DATA=sesug.batchhead N LABEL NOBS;
RUN;
```

Output-

PROC PRINT: List BATCHEAD With Labels

Batch Number	Product Number	Customer ID	Quantity Requested	Unit of Measure	Number of Materials Used
B10000	P0512	HERO	350.00	LB	5
B10010	P0430	HERO	100.00	LB	3
B10020	P0430	DUKES	150.00	LB	3
B10040	P1022	EYESON	200.00	LB	2B

N = 4

Fig.2 Conventional SAS Code, with Output - Listing a Table with Labelled Columns

Figure 2 shows an alternate SAS route to the same results, using **PROC PRINT**. In this example, conventional SAS and SQL are similar in complexity. Watch for later examples which show pronounced differences.

Next, A Mixer

We often want to print a list of Batch Headers with their matching detail records for historical purposes, to track how much of which lots of Raw Material went into each batch of product. Figure 3 illustrates the use of multiple sources - or **JOINS** - and the **WHERE** clause.

The **FROM** clause in this example joins 2 sources, **batchhead** and **batchdet**. After each table name is a

nickname for that table - *sesug.batchhead* is called *head* and *sesug.batchdet* is *det*. These aliases are needed for the **SELECT** clause, which here enumerates each column or field we want to see. Since most of the column names are found only in one table, they need no introduction. However one column, *btnum* (Batch Number) is found in both tables. Therefore we need to indicate that we want to retrieve the *btnum* in *sesug.batchhead* by adding the alias *head* as a prefix. This table.column (here alias.column) syntax is standard SQL. Since there is no provision for a libref in this syntax, we need to use the alias. If we were taking our data from a temporary data set, we could use the whole data set name as the prefix, since the "work" libref is assumed.

SQL Code-

```
TITLE1 'PROC SQL: Combine and Print BATCHEAD With
BATCDET';
PROC SQL;
    SELECT head.btnum, custid, matseq, rnum, lotnum, qtyused
    FROM sesug.batchhead head, sesug.batchdet det
    WHERE head.btnum = det.btnum
;
QUIT;
```

Output-

PROC SQL: Combine and Print BATCHEAD With BATCDET

Batch Number	Customer ID	Material Sequence #	Raw Material Number	Lot Number	Qty Used
B10000	HERO	1	Talc	L100	200.0000
B10000	HERO	2	Fragrance	L101	30.0000
B10000	HERO	3	Blue #12	L100	31.0000
B10000	HERO	4	Lanolin	L120	150.0000
B10000	HERO	5	Carmine	L110	50.0000
B10010	HERO	1	Talc	L101	100.0000
B10010	HERO	2	Carmine	L110	78.0000
B10010	HERO	3	Fragrance	L101	30.0000
B10020	DUKES	1	Talc	L101	150.0000
B10020	DUKES	2	Carmine	L110	117.0000
B10020	DUKES	3	Fragrance	L101	45.0000
B10040	EYESON	1	Lanolin	L120	200.0000
B10040	EYESON	2	Fragrance	L101	20.0000

Fig - 3

The **WHERE** clause states that we want only a subset of the items selected from the sources. In this case, we want only header/detail combinations in which both *batchhead* and *batchdet* have the same batch number.

The equivalent old SAS approach to this task, shown in Figure 4, involves two steps, a **DATA** step **MERGE** followed by a **PROC PRINT**. Since SAS must process the data two times, this technique may be slower on larger files. The fact that it creates a temporary data set may also cause complications such as disk space shortages and leftover files. Take note of the fact that while both routines list the fields to be processed, in the **SQL SELECT** clause the list must be comma-delimited while in the **PROC PRINT VAR** statement there are no commas. This is typical. ALL items in an SQL list must be separated by commas while only occasionally does conventional SAS require commas (or even allow them).

SAS Code-

```
TITLE1 'DATA STEP: Combine and Print BATCHEAD With
BATCDET';
DATA temp;
    MERGE sesug.batchhead sesug.batchdet;
    BY btnum;
RUN;
PROC PRINT DATA=temp LABEL NOOBS;
    VAR btnum custid matseq rnum lotnum qtyused;
RUN;
```

Output-

DATA STEP: Combine and Print BATCHEAD With BATCDET

Batch Number	Customer ID	Material Sqnce #	Raw Material Number	Lot Number	Quantity Used
B10000	HERO	1	Talc	L100	200.0000
B10000	HERO	3	Blue #12	L100	31.0000
B10000	HERO	4	Lanolin	L120	150.0000
B10000	HERO	5	Carmine	L110	50.0000
B10010	HERO	1	Talc	L101	100.0000
B10010	HERO	2	Carmine	L110	78.0000
B10010	HERO	3	Fragrance	L101	30.0000
B10020	DUKES	1	Talc	L101	150.0000
B10020	DUKES	2	Carmine	L110	117.0000
B10020	DUKES	3	Fragrance	L101	45.0000
B10040	EYESON	1	Lanolin	L120	200.0000
B10040	EYESON	2	Fragrance	L101	20.0000

Fig - 4

Now, For Something New

Sometimes we need to figure out how much money we have tied up in Raw Material inventory. To do so, we take the Quantity on hand in each Lot of each Raw Material, and multiply it by the Unit Price for the Lot. Since our accounting department also requests this info for its own analysis, we store the computed values in a new SAS data set, organized by material. Figure 5 shows an SQL way to do this. The **CREATE TABLE** statement is used to make a new data set in SQL. Since the new table is to be made from an existing data set, **CREATE TABLE** is followed by **AS** and a **SELECT** clause to extract our data.

The **SELECT** clause here is notable because it contains a non-existent field. SQL creates the current value, or *valnow* field on the fly by multiplying the Unit Price for the material in each Lot by the current Quantity on hand. While this method of creating a computed variable is standard for SQL, the **FORMAT** and **LABEL** clauses I included are pure SAS. Please note that any standard SAS attributes could have been inserted here as part of the variable name list. If the name of this new variable were followed by another field name, there would be a comma before the next variable name. Many SQL implementations have similar formatting features, but they are not exactly the same.

Another feature introduced here is the ability to sort output with the **ORDER BY** clause. **ORDER BY** can be used to sort by any of the selected variables.

SQL Code-

```
TITLE1 'PROC SQL: Make a Lot Table With Current Value of Lot';
TITLE2 'Sorted By Material';
PROC SQL;
  CREATE TABLE work.lotval AS
  SELECT rmmum, lotnum, qtynow, unitprc,
    qtynow*unitprc AS valnow FORMAT 10.2 LABEL 'Current
    Value'
  FROM sesug.rmlot
  ORDER BY rmmum
  ;
  SELECT * FROM work.lotval;
QUIT;
```

Output-

PROC SQL: Make a Lot Table With Current Value of Lot'
Sorted By Material

Raw Material#	LotNumber	Quantity OnHand	Unit Price	Current Value
Blue #12	L100	100.00	2.00	200.00
Carmine	L110	1556.00	3.00	4668.00
Carmine	L111	1000.00	2.80	2800.00
Fragrance	L100	0.00	5.25	0.00
Fragrance	L101	199.00	5.00	995.00
Lanolin	L120	234.00	1.00	234.00
Lanolin	L119	0.00	1.00	0.00
Lanolin	L121	1500.00	1.00	1500.00
Talc	L102	3000.00	1.50	4500.00
Talc	L101	2000.00	1.50	3000.00
Talc	L100	18.00	1.48	26.64

Fig. 5 Make a New Table Including Computed Fields, Sorting, Multiple Statements in PROC SQL

This example included two SQL requests. The first, already discussed, creates the new data set from the old *rmlot*. The second is the familiar **SELECT ***, used here to print the contents of the new table. Here is a case where we might combine **PROC SQL** with conventional SAS, using SQL to create the new table and **PROC PRINT** to print it. Figure 6 shows a **PROC PRINT** that could be inserted after the SQL in Figure 5 to produce a list that subtotals by Raw Material.

SAS Code-

```
TITLE1 'PROC PRINT: List and Subtotal LOTVAL Created by SQL';
PROC PRINT DATA=work.lotval LABEL NOOBS;
  ID rmmum;
  BY rmmum;
  SUM qtynow valnow;
RUN;
```

Output-

PROC PRINT: List and Subtotal LOTVAL Created by SQL

Raw Material #	Lot Number	Quantity On Hand	Unit Price	Current Value
Blue #12	L100	100.00	2.00	200.00
Carmine	L110	1556.00	3.00	4668.00
	L111	1000.00	2.80	2800.00
Carmine		2556.00		7468.00
		etc....etc....etc....etc....etc....etc....etc		
		9607.00		17923.64

Fig. 6 PROC PRINT to List Table Produced With PROC SQL

Just Looking

It is useful for us to be able to look at a whole batch, including header info, list of ingredients, and raw material cost. However since we have many batches, a data set combining all this information might take too much space. And since our data base is updated constantly by many operators, a table that combined this data would usually be out-of-date. With SQL, we can create a **VIEW** that gives us the data we want and that is always current without tying up all that storage. Figure 7 creates a **VIEW** that assembles the info we need for a batch report and then prints one of the batches using **PROC PRINT**.

The **CREATE VIEW** statement used here has the same syntax as the **CREATE TABLE** in an earlier example. The big difference is that the result of the **CREATE TABLE** is a static data set containing data from pre-existing tables. The **CREATE VIEW** on the other hand produces a small module containing only the instructions for extracting data from the old tables. It contains no actual data. When you use the **VIEW**, as in the **PROC PRINT** in Figure 7, the stored **SELECT** statement is exercised on the current content of the underlying tables and your request is filled. The next time you use the **VIEW**, maybe in a **DATA** step **SET** statement, the request will include the then-current contents of the source tables. Some of the advantages to using a **VIEW** instead of a data set for combining information are reduced storage requirements, up-to-the-minute data, and the ability to change the fields included without actually moving any data. On the negative side, a **VIEW** with lots of data may take longer to access, because the virtual table must be created every time you open it. Also, you may want a static snapshot of your underlying data, and the **VIEW** will not provide that. For example, if the underlying tables are not available or their layout changes, your **VIEW** will not work. In addition, SAS data **VIEWS** offer only read access to your data. You cannot update them.

You can see that the **SELECT** clause used to create this **VIEW** has the same features as those used with tables. It includes table alias prefixes on many of the column names to specify the sources of the columns and the ***** key phrase to get all the columns in the *batchhead* table. It specifies a computed variable for total raw material prices complete with **SASFORMAT** and **LABEL**.

The **FROM** clause covers three tables this time, each with an alias. The **DROP=** term (with *batchhead*) is a SAS data set option. You can use most data set options (e.g. **DROP**, **KEEP**, **RENAME**) with **PROC SQL** in **CREATE TABLE** and **CREATE VIEW** clauses as well as in **FROM**. This is obviously a SAS extension - a very useful one. In Figure 7 we wanted to retrieve all the variables from *batchhead* except *nummat*. We **SELECTed** the key

phrase * for all variables and then excluded *nummat* from consideration with the **DROP** parameter.

SQL and Other SAS Code-

TITLE1 'PROC SQL: Create a View For Reviewing Batch Costs';

PROC SQL;

CREATE VIEW sesug.batcost AS

SELECT head.*,

det.rmnum, det.lotnum, matseq, qtyused,

det.unitmeas,

det.qtyused*lot.unitpric AS usedpric FORMAT

10.2 LABEL 'R.M. Cost', lot.unitpric,

FROM sesug.batchhead(DROP=nummat) head,

sesug.batchdet det, sesug.rmlot lot

WHERE head.btnum=det.btnum AND

det.rmnum=lot.rmnum AND

det.lotnum=lot.lotnum

ORDER BY btnum, matseq

QUIT;

TITLE1 'PROC PRINT: List Batch Costs From View';

PROC PRINT

DATA=sesug.batcost(WHERE=(btnum='B10010'))

LABEL NOOBS;

ID bttnum prodnum qtyreqst;

BY bttnum prodnum qtyreqst;

SUMBY bttnum;

VAR rmnum lotnum qtyused unitpric;

SUM usedpric;

RUN;

Output-

PROC PRINT: List A Batch's Costs From View

Raw							
Batch	Product	Qty	Mtrl	Lot	Qty	Unit	RM
#	#	Rqstd	#	#	Used	Price	Cost
B10010	P0430	100	Talc	L101	100	1.50	150.00
			Carmine	L110	78	3.00	234.00
			Fragrance	L101	30	5.00	150.00
							=====
B10010							534.00
							=====
							534.00
							=====
							2443.00

Fig. 7 Create View From Three Tables and Print With Data Set WHERE Option

The **WHERE** and **ORDER BY** clauses in the **PROC SQL** are as described earlier in this paper. There is another **WHERE** however, in the **PROC PRINT DATA=** clause. It is used to print only a single batch. This kind of **WHERE** is another SAS data set option, which can be used with any reference to an input data set or view (e.g. **SET**, **MERGE** statement, **DATA=** clause). It differs from the **SQL WHERE** clause in its layout and limitations. The comparison part must be in parenthesis after **WHERE=**. It can refer only to other columns in the same observation or to constants. It cannot, for instance, be used to select the batch with the highest cost or the batch that used a raw material no longer in stock. An **SQL WHERE** clause

could do those things. Check the **SQL** and **SAS Language** documentation for details on other minor differences.

Outer Limits

Another regular request in this system is for the disposition of Lots of Raw Material. We need to know how much material we received and which batches used it. A regular **JOIN** (also called an **INNER JOIN**) such as those we have been doing retrieves only those records that have matching values in all source data sets. In this case we want to list all lots, even those that have not yet been used. Figure 8 shows how to perform this task with an **OUTER JOIN**. The idea of an **OUTER JOIN** is to retrieve all the rows from the primary table and merge them with whatever data is available from another table.

SQL Code-

TITLE1 'PROC SQL: List all Raw Material Lots';

TITLE2 'With Any Batches That Used Them';

PROC SQL;

SELECT lot.rmnum, lot.lotnum, btnum, qtyrecd,

lot.unitmeas, qtyused

FROM sesug.rmlot lot LEFT JOIN sesug.batchdet det

ON lot.rmnum=det.rmnum AND

lot.lotnum=det.lotnum

ORDER BY rmnum, lotnum

;

%PUT Return Code=&SQLRC

Number of Obs=&SQLOBS;

QUIT;

Output-

Log Message: Return Code=0 Number of Obs=18

PROC SQL: List all Raw Material Lots
With Any Batches That Used Them

Raw Mtrl	Lot #	Batch #	Qty Rcvd	Unit of Measure	Qty Used
Blue#12	L100	B10000	100	GM	31.00
Carmine	L110	B10020	2000	GM	117.00
Carmine	L110	B10000	2000	GM	50.00
Carmine	L110	B10010	2000	GM	78.00
Carmine	L111		1000	GM	
Fragrance	L100		500	OZ	
Fragrance	L101	B10020	350	OZ	45.00
Fragrance	L101	B10010	350	OZ	30.00
Fragrance	L101	B10040	350	OZ	20.00
Fragrance	L101	B10000	350	OZ	30.00
Lanolin	L119		1500	LB	
Lanolin	L120	B10040	1500	LB	200.00
Lanolin	L120	B10000	1500	LB	150.00
Lanolin	L121		1500	LB	
Talc	L100	B10000	2000	LB	200.00
Talc	L101	B10020	2000	LB	150.00
Talc	L101	B10010	2000	LB	100.00
Talc	L102		3000	LB	

Fig. 8 PROC SQL: Left Outer Join to List Even Non-Matching Records, Showing Automatic Macro Variables.

In Figure 8, following an unremarkable **SELECT** clause is a new kind of **FROM** clause, which specifies how the two source tables will be joined. In this case, the term **LEFT JOIN** replaces the usual comma between table names and the **ON** clause replaces **WHERE**. **LEFT JOIN** signifies that the table listed on the left in the clause (*sesug.rmlot*) is dominant. That is, data from every row in the left table is included in the output but only those rows in the right table (*sesug.batchdef*) that match rows in the left will be included. The effect is like that of performing a **DATA** step **MERGE** where only observations containing data from a specific data set are processed. If the **FROM** clause specified **RIGHT JOIN**, the table on the right would be dominant. **FULL JOIN** would state that all rows, matching or not, are to be included. **ON** is used instead of **WHERE** with **JOIN** terms. An **OUTER JOIN** is a standard SQL concept, but check your syntax before trying it in some other package. It will probably be different.

Note the **MACRO %PUT** statement before the **SQL QUIT**. A valuable **PROC SQL** feature is the automatic **MACRO** variables it sets. After every **SQL** statement (within a **PROC SQL**) SAS puts a completion code into **SQLRC** (0 for normal completion) and the number of observations retrieved into **SQLOBS**. By checking these two Macro variables after a query you can conditionally execute later steps. For instance, if a query had a bad return code (**&SQLRC NE 0**), you might print an error message.

Conclusion

PROC SQL provides an alternative method for processing data in SAS. Its most common advantage is that its syntax is standardized with other packages and data base. For some tasks, it is also just the best way to go.