

PROC SQL - A SAS® Programmer's Perspective

Deborah L. Skinner

John Alden Life Insurance Company, Sacramento, CA

PROC SQL was introduced starting with Version 6.06 of the SAS® system. Prior to its introduction, SAS users have traditionally used

- SET, MERGE or UPDATE in DATA steps to merge files
- PROC SORT to sort files
- PROC PRINT to create simple reports
- PROC MEANS, PROC FREQ, or PROC SUMMARY to summarize data.

PROC SQL contains functionality from all of these and is a powerful tool in the SAS programmer's toolkit. It will not replace the traditional programming methods, but it will enhance your access to your data and make you a more productive programmer.

The purpose of this paper is to compare traditional SAS methods to SQL methods for performing a few common programming tips. Since computer performance varies across platforms, we will look at performance in terms of human efficiency - that is, the simplicity of the code and the ease of maintenance.

Why Learn and Use SQL?

You may be wondering just why it is that you should learn another programming method. After all, the way you've been doing things works just fine and you are able to accomplish everything you need to get done.

- SQL can often accomplish a task in either fewer or simpler steps than a traditional SAS® approach. This is especially true in fuzzy merges - but may also be true if you need to create a new variable, then sort the data, and then print a report. While you could accomplish this with a DATA step, PROC SORT and PROC PRINT, PROC SQL will accomplish this in one step instead of three.

- SQL crosses the line into other programming languages and applications. While you may have

some difficulty communicating with your Oracle developer using SAS® terminology, you might find it to be a lot easier if you can "speak" SQL.

- Knowledge of SQL may make you more attractive in the job market. While SAS® SQL, like all varieties, has its own extensions, it is adherent to the ANSI standard for all SQL's.

- Many shops place fairly limited resource constraints on SAS®, while allocating larger chunks to RDMS stores. You may be able to make more effective use of your employer's RDMS and your organization's computing power by using SQL Pass-Through and SAS/ACCESS®.

- You may be able to leverage existing your organization's existing RDMS logic by incorporating it into PROC SQL with pass-through and then using the resulting data set for analysis and reporting.

Why not use it all the time?

There are some things that SQL does not do particularly well - or not at all. Here are some tasks that you will want to reserve for traditional methods.

- If you need multiple output data sets, SQL will not create multiple tables from one step.
- If you have multiple matching or no-match criteria, SQL may not handle it.
- If your application requires the use of the LAGn function. While SQL supports nearly all SAS® functions, LAGn is the notable exception.
- If you need sophisticated control of data handling or your program contains complex logic.
- If you need a report where you have a lot of control over the layout and appearance.

SQL - A New Vocabulary

Before we get started on the comparisons of traditional SAS® methods to SQL, it is important to know a little about the SQL vocabulary. Discussions

about data base normalization, entities, primary and foreign keys are beyond the scope of this paper. Terms you should be familiar with are:

SQL	Traditional SAS
Table	Data set
Row	Observation
Column	Variable

These terms will be used interchangeably during the course of this paper.

Getting Started - Traps for the SAS® Programmer to Avoid

Punctuation in SQL will seem a little strange at first. Some of us even sign our names with a semi-colon.

- In SQL, there is only one semi-colon. It appears at the very end of a SQL statement.

- SQL uses commas to separate row and table names.

Unless you explicitly code around it, SQL will return both occurrences of common variables when you join tables. This is a marked contrast to the SAS® MERGE statement where common variables are overwritten in the order the tables are merged.

You also don't need to use DROP= and KEEP= processing. SQL will perform it's equivalent based on the items in your SQL select and where clauses.

Tables (Data Sets) to be Used in Examples

For purposes of the examples to follow, I will be using a simplified version of the types of data I work with every day - that is, group health insurance data.

The tables are:

Data Set Name: SASDATA.CLAIMS

-----Alphabetic List of Variables and Attributes-----

#	Variable	Type	Len	Pos	Format
4	BILLED	Num	8	13	12.2
7	CLM_PAID	Num	8	37	DATE7.
6	CLM_REC'D	Num	8	29	DATE7.
3	CPT_CODE	Char	5	8	
1	INSUR_ID	Char	4	0	
2	PROVD_ID	Char	4	4	
5	SERV_DT	Num	8	21	DATE7.

This table represents a set of medical services that the insurance company has received a claim for.

Data Set Name: SASDATA.INSURED

-----Alphabetic List of Variables and Attributes-----

#	Variable	Type	Len	Pos	Format
4	COV_CNCL	Num	8	16	DATE7.
3	COV_EFF	Num	8	8	DATE7.
5	COV_TYPE	Char	3	24	
2	EMP_ID	Char	4	4	
1	INSUR_ID	Char	4	0	
6	INSUR_NM	Char	25	27	

This table contains information about the people that the insurance company insures.

Data Set Name: SASDATA.PROVIDER

-----Alphabetic List of Variables and Attributes-----

#	Variable	Type	Len	Pos
1	PROVD_ID	Char	4	0
2	PROVD_NM	Char	25	4
3	PROVD_ST	Char	2	29

This table contains information about the providers of medical services, typically doctors or hospitals.

Data Set Name: SASDATA.EMPLOYER

-----Alphabetic List of Variables and Attributes-----

#	Variable	Type	Len	Pos
1	EMP_ID	Char	4	0
2	EMP_NAME	Char	25	4
3	EMP_STAT	Char	2	29

And this table contains information about the employers that the insured people work for. (Typically group health insurance is a employer provided benefit.)

Example 1: Create a new column, summarize the data, sort the data, and print a report.

Suppose we want to generate a report showing the average number of days from the time a medical service is provided until the claim is received by the insurance company. We want the report grouped by the provider of medical services in descending order by the average length of time.

Using traditional SAS® programming methods, you may approach the problem like this:

```
data claims (keep=provd_id days);
  set sasdata.claims (keep=provd_id clm_rec'd
serv_dt);
  days=clm_rec'd-serv_dt;
run;
```

```
proc summary nway data=claims;
  class provd_id;
  var days;
  output out=provsumm mean=;
```

```
run;

proc sort data=sasdata.provider out=provider;
  by provd_id;
run;

data provsumm;
  merge provsumm provider;
  by provd_id;
run;

proc sort data=provsumm;
  by descending days;
run;

title 'Traditional SAS Programming Method Report';

proc print data=provsumm;
  id provd_nm;
  var days;
run;
```

This approach creates the variable “days” in a data step, summarizes the days by provider in a PROC SUMMARY, merges that data with the data set containing the provider’s name, once again sorts the data by the descending average days, and then uses PROC PRINT to produce the simple report.

The output produced looks like this:

```
Traditional SAS Programming Method Report

PROVD_NM          DAYS
Dr. Bob Steele    195.00
Cornerstone Medical Group  23.25
Oak Street Medical Clinic  22.80
```

This method takes two DATA steps and four PROC steps. But using PROC SQL, you can accomplish the same result with one step.

Here is how the same thing is done in PROC SQL:

```
title 'PROC SQL Report';

proc sql;
select
  provider.provd_nm
  ,avg(claims.clm_recd - claims.serv_dt)
    as avedays
from sasdata.provider
  ,sasdata.claims
where provider.provd_id=claims.provd_id
group by provider.provd_nm
order by avedays desc;
```

The report that is produced looks nearly identical to the one that we got out of the six step process.

PROC SQL Report

PROVD_NM	AVEDAYS
Dr. Bob Steele	195
Cornerstone Medical Group	23.25
Oak Street Medical Clinic	22.8

Let’s take a look at the SQL code and see exactly what each step does:

```
proc sql;
```

- This is how you tell SAS® that the statements that follow are SQL statements. The SQL processor is invoked and control is taken away from the traditional DATA and PROC compiler.

```
select
  provider.provd_nm
  ,avg(claims.clm_recd - claims.serv_dt)
    as avedays
```

- The select clause is extremely powerful. With this statement, we are
 - creating a new column (avedays)
 - defining the order the columns will appear on the report (first provider name, then average days)
 - defining the summarization statistic to be used on the average days. This is used in conjunction with the group by clause later in the SQL.

```
from sasdata.provider
  ,sasdata.claims
```

- The from clause defines which tables (data sets) are being used in the report. Note that they do not need to be pre-sorted in order to be joined or merged.

```
where provider.provd_id=claims.provd_id
```

- The where clause defines how the two tables are to be joined. In the absence of a where clause, SQL creates a Cartesian product join, where every observation of the first table would be joined with every observation of the second.

```
group by provider.provd_nm
```

- The group by clause operates in conjunction with the aggregation operator specified in the select clause. Whenever you have aggregation you need to consider a group by clause so that SQL knows how to group the data. In the absence of a group by, the result will be the aggregation of the entire table.

```
order by avedays desc;
```

- The order by clause determines the sort order of the final report. And because it is the last of the SQL clauses, it is followed by the lone semi-colon.

Pretty easy, eh? Let's look at another example.

Example 2: "Fuzzy" Merge

Whenever you are dealing with date-driven data - very common in insurance and financial applications - you may not always have an exact variable to join by. Instead, as in this example, you may need to check to see if a date from one file is between two dates on another file.

Let's create a report showing the type of coverage that the insured person has as of the day the medical services are provided. The type of coverage is on the insured table and is bounded by an effective and cancellation date. We need to match the insured table with the claims table and determine the coverage in effect on the medical service date.

Using traditional SAS® programming methods, we would probably do something like this:

```
data claimcov;
  set sasdata.claims
  (rename=(insur_id=clins_id));
  do x=1 to last by 1;
    set sasdata.insured nobs=last point=x;
    if insur_id=clins_id and
      ((cov_eff <= serv_dt <= cov_cncl
        or cov_eff <= serv_dt and
        cov_cncl=.)
      then output;
    end;
  run;

proc sort data=claimcov;
  by insur_nm;
run;

title 'Traditional SAS Programming Report';

proc print;
  id insur_nm;
  var cpt_code serv_dt cov_type;
run;
```

To solve this problem, we could use a "set-set" solution in the DATA step, where we check every observation in the insured data set against every observation in the claims data set and output those where the insured identifiers match up and the service was provided between the coverage effective date and the coverage cancellation date.

Note: The purpose for the secondary logic is that the cancellation date is missing on those records where the coverage remains active. Therefore, we need to check both between the dates and for the situation where the services are provided after the coverage effective date and the cancel date is missing.

Once we have done that, we can sort the data and print the report, producing a report that looks like this:

Traditional SAS Programming Report

INSUR_NM	CPT_CODE	SERV_DT	COV_TYPE
Bill Jones	99211	16NOV95	MED
Bill Jones	99070	23NOV95	MED
Bill Jones	87081	18NOV95	MED
Bill Jones	71010	18OCT95	MED
Bill Jones	97112	07MAR96	MED
Jan Martin	87210	18NOV95	MM2
Lilly Brown	70459	01JUN96	MED
Margaret Smith	99212	01DEC95	MED
Ron North	88302	26MAY95	MM1
Tom Kent	97110	17NOV95	MM1

The fact that we are creating an all-to-all merge in this example is a clue that it is a natural for PROC SQL. PROC SQL creates a Cartesian product (all-to-all merge) and then applies the where clause to it. So the one-step PROC SQL solution to the same problem is:

```
title 'PROC SQL Report';

proc sql;
  select
    insured.insur_nm
    ,claims.cpt_code
    ,claims.serv_dt
    ,insured.cov_type
  from
    sasdata.claims
    ,sasdata.insured
  where insured.insur_id=claims.insur_id
    and ((insured.cov_eff <= serv_dt <=
    cov_cncl)
    or insured.cov_eff <= serv_dt and
    cov_cncl=.)
  order by insured.insur_nm
;
```

PROC SQL Report

INSUR_NM	CPT_CODE	SERV_DT	COV_TYPE
Bill Jones	99211	16NOV95	MED
Bill Jones	71010	18OCT95	MED
Bill Jones	99070	23NOV95	MED
Bill Jones	97112	07MAR96	MED
Bill Jones	87081	18NOV95	MED
Jan Martin	87210	18NOV95	MM2
Lilly Brown	70459	01JUN96	MED

Margaret Smith	99212	01DEC95	MED
Ron North	88302	26MAY95	MM1
Tom Kent	97110	17NOV95	MM1

Example 3: First-dot and Last-dot processing can sometimes be done with SQL.

First-dot and last-dot programming has always been one of the most useful tricks in my SAS® repertoire. When I first started using SQL, I thought that one of the weaknesses of SQL was that it couldn't accommodate this type of processing. But depending on your application, this may or may not be true.

Here is an example where SQL performs the equivalent of last-dot processing.

The report we are producing in this example is the latest date that each insured person has received medical services. It is to include the insured's name and the date of their most recent medical service.

```
proc sort data=sasdata.claims out=claims;
  by insur_id serv_dt;
run;

data claims;
  set claims (keep=insur_id serv_dt);
  by insur_id;
  if last.insur_id;
run;

proc sort data=sasdata.insured out=insured;
  by insur_id;
run;

data claims;
  merge claims
        insured;
  by insur_id;
run;

title 'Traditional SAS Programming Report';

proc print;
  id insur_id;
  var insur_nm serv_dt;
run;
```

Using the traditional SAS programming approach, we would sort the claims by insured and service date. Then in the data step, we would take the last-dot observation for each insured because the sort would have made it the most recent. We would then sort that result by insured identifier so that we can merge it with the insured data set containing the insured name, and then print the report.

The report that is produced is:

Traditional SAS Programming Report

INSUR_ID	INSUR_NM	SERV_DT
1234	Bill Jones	07MAR96
1236	Lilly Brown	01JUN96
1238	Tom Kent	17NOV95
1239	John Miller	19JUN96
1245	Margaret Smith	01DEC95
1295	Jan Martin	18NOV95
1560	Ron North	26MAY95

By using a little SQL "trick" we can accomplish the same end. Here is the SQL solution:

```
title 'PROC SQL Report';
proc sql;
  select
    claims.insur_id
    ,insured.insur_nm
    ,max(serv_dt) as lastdate format=date7.
  from sasdata.claims
    ,sasdata.insured
  where claims.insur_id=insured.insur_id
  group by claims.insur_id
    ,insured.insur_nm
  order by claims.insur_id
    ,insured.insur_nm;
```

The trick here is the MAX aggregation function. We use the group by clause in conjunction with the aggregation to get the most recent date.

PROC SQL Report

INSUR_ID	INSUR_NM	LASTDATE
1234	Bill Jones	07MAR96
1236	Lilly Brown	01JUN96
1238	Tom Kent	17NOV95
1239	John Miller	19JUN96
1245	Margaret Smith	01DEC95
1295	Jan Martin	18NOV95
1560	Ron North	26MAY95

Example 4: The three-way merge

A three-way merge occurs when two tables are not directly related to each other, but instead are related indirectly through a third table. An example of this would be the claims table and the employer table can be related to each other through the insured table.

The report we want to produce for this example lists the employer's name, the insured's name and the total billed charges for this insured. Using the traditional SAS® approach, we can accumulate the billed charges by insured, merge that with the insured data set to get the insured's name, and then

sort by employer to merge with the employer data set.

Here is the traditional SAS® code:

```
proc sort data=sasdata.claims out=claims;
  by insur_id;
run;

data clmsumm;
  set claims;
  by insur_id;
  retain totbill;
  totbill + billed;
  if last.insur_id then do;
    output;
    totbill=0;
  end;
run;

proc sort data=sasdata.insured out=insured;
  by insur_id;
run;

data ins_clms;
  merge insured
        clmsumm;
  by insur_id;
run;

proc sort data=ins_clms;
  by emp_id;
run;

proc sort data=sasdata.employer out=employer;
  by emp_id;
run;

data emp_clms;
  merge ins_clms
        employer;
  by emp_id;

title 'Traditional SAS Programming Report';

proc print data=emp_clms;
  id emp_id;
  var emp_name insur_nm totbill;
run;
```

The report that is produced follows:

Traditional SAS Programming Report

EMP_ID	EMP_NAME	INSUR_NM	TOTBILL
6677	Main Street Florist	Bill Jones	95.0
6677	Main Street Florist	Margaret Smith	34.0
6677	Main Street Florist	Ron North	75.0
6678	Bill's Hardware	Lilly Brown	154.5
6678	Bill's Hardware	John Miller	30.5
6679	Joe's Best Burgers	Tom Kent	45.0
6679	Joe's Best Burgers	Jan Martin	18.0

Once again, all of these steps can be accomplished with one PROC SQL step. The SQL code that joins the tables is:

```
title 'PROC SQL Report';
proc sql;
select
  employer.emp_id
  ,employer.emp_name
  ,insured.insur_nm
  ,sum(claims.billed)
from sasdata.employer
  ,sasdata.insured
  ,sasdata.claims
where employer.emp_id=insured.emp_id and
      insured.insur_id=claims.insur_id
group by employer.emp_id
      ,employer.emp_name
      ,insured.insur_nm
;
```

And produces this report:

PROC SQL Report			
EMP_ID	EMP_NAME	INSUR_NM	TOTBILL
6677	Main Street Florist	Bill Jones	95
6677	Main Street Florist	Margaret Smith	34
6677	Main Street Florist	Ron North	75
6678	Bill's Hardware	John Miller	30.5
6678	Bill's Hardware	Lilly Brown	154.5
6679	Joe's Best Burgers	Jan Martin	18
6679	Joe's Best Burgers	Tom Kent	45

Conclusion

This has been a brief overview of how SQL can help you take some of your traditional SAS® code and simplify some simple programming tasks. Standard SAS® is still a convenient and powerful tool for accomplishing your data processing functions. But by using SQL to supplement the strengths of the traditional SAS® approach, you can make your programs simpler and more versatile.

The areas where SQL is strongest seem to be:

- All-to-all (or many-to-many) merges where a Cartesian product subset is required.
- “Fuzzy” merges where the logic requires “between” type merging rather than direct one-to-one merging.
- Under certain circumstances, selection of distinct rows (observations) from tables analogous to the traditional SAS® first-dot and last-dot processing.
- Joining multiple tables when there is not a common key to join them in one step.

For further information, please feel free to contact the author at:

John Alden Life Insurance Company
1610 Arden Way Suite 175
Sacramento, CA
916-565-5898

References

Dickson, Alan and Pass, Ray, "Select Items from PROC.SQL Where Items > Basics", *Proceedings of the Third Annual Southeast SAS® Users Group Conference*, 1995, pp.12-21

Kent, Paul , "A Taste of SQL", *Proceedings of the Third Annual Southeast SAS® Users Group Conference*, 1995, pp. 189-193

Levine, Howard, "Strengths and Weaknesses of the SQL Procedural Language Compared to Standard SAS Language" , *Proceedings of the Eighteenth Annual SAS® Users Group International Conference*, 1993, pp 1417 - 1423

SAS® Institute, *SAS® Guide to the SQL Procedure, Usage and Reference, Version 6, First Edition* , 1989

SAS® is a registered trademark of SAS® Institute Inc. in the USA and other countries. ® indicates USA registration.