ABSTRACT

PROC SQL® is a powerful yet still overlooked tool within our SAS® arsenal. PROC SQL can create tables, sort and summarize data, and join/merge data from multiple tables and in-line views. The SELECT statement with the CASE-WHEN clause can conditionally process the data like the IF-THEN-ELSE statement in the DATA step. An advantage specific to PROC SQL is that with careful coding, the SQL code can be ported to 3rd party Relational Database Management Systems (RBMS) such as Oracle® and SQL Server® with virtually no changes. This paper will show some techniques to QA and reshape data the way you want to see it, with a focus on in-line views.

INTRODUCTION

Structured Query Language (SQL) was originally designed to extract data out of a relational database but we can also use it to process SAS data files. SQL has many flavors depending on the relational database system and most flavors follow similar syntax so the same basic code can be used across the different flavors of SQL. This basic syntax is called ANSI SQL. The SAS implementation of SQL is in PROC SQL and enables the SAS programmer to leverage this powerful tool into code that is easier to follow and is more efficient.

This paper will have the reader follow the journey of an analyst working for a fictitious international shoe company that needs to create a series of adhoc management reports that will (unbeknown to our analyst) eventually be ported to the company’s production database. Our journey will lead from simple listings to complex conditional processing.

EXAMPLE – SASHHELP.SHOES

Since every installation of SAS comes with sample data in the SASHELP library, the data file SASHHELP.SHOES will be used to demonstrate the PROC SQL code used throughout this paper. The SHOES data file represents the sales and inventory data for a fictitious shoe company with stores worldwide. Our job as an analyst is to analyze the data and generate summary reports to management. But first let us see the contents of the data file. As with PROC DATASETS and PROC CONTENTS, PROC SQL we can get a layout of the data file with the DESCRIBE statement.

```
proc sql;
  describe table sashelp.shoes;
quit;
```

The output of the DESCRIBE statement occurs in the SAS log. Below is an excerpt of the result.

```
create table SASHHELP.SHOES( label='Fictitious Shoe Company Data' bufsize=8192 )
  (  
    Region char(25),
    Product char(14),
    Subsidiary char(12),
    Stores num label='Number of Stores',
    Sales num format=DOLLAR12. informat=DOLLAR12. label='Total Sales',
    Inventory num format=DOLLAR12. informat=DOLLAR12. label='Total Inventory',
    Returns num format=DOLLAR12. informat=DOLLAR12. label='Total Returns'
  );
```

We are going to quickly review the data by viewing the first 5 rows of data. The OUT0BS option in the PROC SQL statement below selects the top 5 rows out of 395 rows. Note: the OUTOBS= option restricts the displayed output not the number of rows processed.

```
proc sql outobs=5;
  select *
    from sashelp.shoes;
quit;
```
AGGREGATING DATA USING PROC SQL

PROC SQL can of course do more than just list data, PROC SQL can also summarize or aggregate data. Suppose management wants a sales summary for each region. Without PROC SQL, we could accomplish this with PROC MEANS.

```
proc means data=sashelp.shoes sum;
   class region;
   var sales;
run;
```

A similar output can be accomplished with PROC SQL. In the example below, the GROUP BY clause tells PROC SQL to calculate the sum of the sales for each region. In addition to calculating the group level statistics, PROC SQL also can format the display by using any SAS format, something PROC MEANS cannot do. To make the output more readable the format DOLLARw. was used for the summarized sales figure.

```
proc sql;
   select region
       ,count(sales)   as n_obs
       ,sum(sales) as sales  format=dollar16.
   from sashelp.shoes
   group by region;
quit;
```
Figure 3: Summarized Sales by Region – PROC SQL Output

<table>
<thead>
<tr>
<th>Region</th>
<th>n_obs</th>
<th>sales</th>
</tr>
</thead>
<tbody>
<tr>
<td>Africa</td>
<td>56</td>
<td>$2,342,588</td>
</tr>
<tr>
<td>Asia</td>
<td>14</td>
<td>$460,231</td>
</tr>
<tr>
<td>Canada</td>
<td>37</td>
<td>$4,259,712</td>
</tr>
<tr>
<td>Central America/Caribbean</td>
<td>32</td>
<td>$3,657,753</td>
</tr>
<tr>
<td>Eastern Europe</td>
<td>31</td>
<td>$2,394,940</td>
</tr>
<tr>
<td>Middle East</td>
<td>24</td>
<td>$5,631,779</td>
</tr>
<tr>
<td>Pacific</td>
<td>45</td>
<td>$2,296,794</td>
</tr>
<tr>
<td>South America</td>
<td>54</td>
<td>$2,434,783</td>
</tr>
<tr>
<td>United States</td>
<td>40</td>
<td>$5,503,986</td>
</tr>
<tr>
<td>Western Europe</td>
<td>62</td>
<td>$4,873,000</td>
</tr>
</tbody>
</table>

Besides having similar summarizing characteristics as PROC MEANS, PROC SQL can also summarize data based on 2 or more fields. Suppose management wants a report that shows net sales (Sales – Returns) for each region (See figure 4). To do that without PROC SQL, we would have to make another data set with a new variable such as Net_Sales=Sales-Returns, then run PROC MEANS on that new variable. The PROC SQL solution is elegant and will open up a new world of coding to our analyst.

```sql
proc sql;
  select region ,
       sum(sales) as sales format=dollar16.
       ,sum(sales-returns) as net_sales format=dollar16.
  from sashelp.shoes
  group by region;
quit;
```

Figure 4: Summarized Region Sales and Net Sales

<table>
<thead>
<tr>
<th>Region</th>
<th>Sales</th>
<th>Net Sales</th>
</tr>
</thead>
<tbody>
<tr>
<td>Africa</td>
<td>$2,342,588</td>
<td>$2,263,501</td>
</tr>
<tr>
<td>Asia</td>
<td>$460,231</td>
<td>$443,336</td>
</tr>
<tr>
<td>Canada</td>
<td>$4,259,712</td>
<td>$4,125,318</td>
</tr>
<tr>
<td>Central America/Caribbean</td>
<td>$3,657,753</td>
<td>$3,530,655</td>
</tr>
<tr>
<td>Eastern Europe</td>
<td>$2,394,940</td>
<td>$2,303,239</td>
</tr>
<tr>
<td>Middle East</td>
<td>$5,631,779</td>
<td>$5,424,899</td>
</tr>
<tr>
<td>Pacific</td>
<td>$2,296,794</td>
<td>$2,213,665</td>
</tr>
<tr>
<td>South America</td>
<td>$2,434,783</td>
<td>$2,331,932</td>
</tr>
<tr>
<td>United States</td>
<td>$5,503,986</td>
<td>$5,316,484</td>
</tr>
<tr>
<td>Western Europe</td>
<td>$4,873,000</td>
<td>$4,703,245</td>
</tr>
</tbody>
</table>

IN-LINE VIEWS

Our journey as an analyst continues, our manager is asking for a copy of the original SHOES data with the summary of sales by region added to it (See figure 5).

Figure 5: SHOES with Region Sales

<table>
<thead>
<tr>
<th>Region</th>
<th>Product</th>
<th>Subsidiary</th>
<th>Number of Stores</th>
<th>Total Sales</th>
<th>Total Inventory</th>
<th>Total Returns</th>
<th>Region Sales</th>
</tr>
</thead>
<tbody>
<tr>
<td>Africa</td>
<td>Boot</td>
<td>Addis Ababa</td>
<td>12</td>
<td>$29,761</td>
<td>$131,821</td>
<td>$759</td>
<td>$2,342,588</td>
</tr>
<tr>
<td>Africa</td>
<td>Men's Casual</td>
<td>Addis Ababa</td>
<td>4</td>
<td>$67,242</td>
<td>$116,036</td>
<td>$2,284</td>
<td>$2,342,588</td>
</tr>
<tr>
<td>Africa</td>
<td>Men's Dress</td>
<td>Addis Ababa</td>
<td>7</td>
<td>$76,793</td>
<td>$136,272</td>
<td>$2,433</td>
<td>$2,342,566</td>
</tr>
<tr>
<td>Africa</td>
<td>Sandal</td>
<td>Addis Ababa</td>
<td>10</td>
<td>$62,819</td>
<td>$204,254</td>
<td>$1,851</td>
<td>$2,342,688</td>
</tr>
<tr>
<td>Africa</td>
<td>Slipper</td>
<td>Addis Ababa</td>
<td>14</td>
<td>$68,641</td>
<td>$279,795</td>
<td>$1,771</td>
<td>$2,342,588</td>
</tr>
</tbody>
</table>
In order to get the above desired results, we first need to create table in which we will calculated REGION_SALES, then join REGION_SALES to SASHELP.SHOES by region. This is a 2 step process and is demonstrated in the following PROC SQL code.

```sql
proc sql;
create table region_sales as
select region
    ,sum(sales) as region_sales
from sashelp.shoes
group by region;
proc sql outobs=5;
select a.*
    ,b.region_sales format=dollar16. label='Region Sales'
from sashelp.shoes a join region_sales b on a.region=b.region;
quit;
```

Instead of using a table as the source of data for your PROC SQL query, you can use a data structured called an in-line view. An in-line view is a nested query that is specified in the FROM clause. An in-line view selects data from one or more tables to produce a temporary in-memory table. This virtual table exists only during the query. The main advantage of using an in-line view is to reduce the complexity of the code. (SAS Certification Prep Guide).

An in-line view is a SELECT statement within a SELECT statement, which we call a nested statement. Nested SELECT statements sound complex but they are not. In our example we are going to join the output of Figure 3 to the SHOES data to get the results in figure 5.

```sql
proc sql;
select a.*
    ,b.region_sales format=dollar16. Label='Region Sales'
from sashelp.shoes a join (select region
    ,sum(sales) as region_sales
from sashelp.shoes
    group by region) b on a.region=b.region;
quit;
```

Let us examine this code in some detail. The code snippet below is the in-line view. Notice how the in-line is enclosed in parentheses. A table like the table in figure 3 is created in memory and then is joined with SHOES to get our final result set which is identical to the output of the 2 step process. In essence we joined the SHOES data with a summarization of itself.

```sql
(select region
    ,sum(sales) as region_sales
from sashelp.shoes
    group by region)
```

**FINDING DUPLICATES**

We can also uncover duplicate rows in the data by using our newly found arsenal of SQL programming tricks. Our example data SASHELP.SHOES is supposed to be unique by REGION, PRODUCT, and SUBSIDIARY. We could test for and output any offending duplicate rows by using PROC SORT, but we can also do our duplicate testing with PROC SQL by using our newly gained knowledge of in-line views and group by aggregation along with the HAVING clause. With PROC SQL, we can filter data before and after aggregation. The HAVING clause enables us to subset the in-line view after the aggregation is complete.

The below PROC SQL code outputs all rows that have duplicate values of REGION, PRODUCT, and SUBSIDIARY in figure 6.
proc sql;
  select a.*
  from sashelp.shoes a join (select region, product, subsidiary, count(sales) as count
                       from sashelp.shoes
                       group by region, product, subsidiary
                       having count(sales)>1) b on a.region=b.region and
                       a.product=b.product and a.subsidiary=b.subsidiary;
quit;

Figure 6: Duplicate Rows

<table>
<thead>
<tr>
<th>Region</th>
<th>Product</th>
<th>Subsidiary</th>
<th>Number of Stores</th>
<th>Total Sales</th>
<th>Total Inventory</th>
<th>Total Returns</th>
</tr>
</thead>
<tbody>
<tr>
<td>Western Europe</td>
<td>Sport Shoe</td>
<td>Copenhagen</td>
<td>1</td>
<td>$1,927</td>
<td>$17,683</td>
<td>$37</td>
</tr>
<tr>
<td>Western Europe</td>
<td>Sport Shoe</td>
<td>Copenhagen</td>
<td>13</td>
<td>$131,922</td>
<td>$327,742</td>
<td>$4,204</td>
</tr>
</tbody>
</table>

The in-line view piece of the PROC SQL code, first generates a count for each REGION, PRODUCT, and SUBSIDIARY. Then the HAVING clause only keeps those rows where the count is more than 1. The HAVING clause must appear after the GROUP BY clause, and is processed after the GROUP BY is completed.

CONDITIONAL PROCESSING WITH CASE-WHEN

Once more our manager comes in and has another data call. Management wants to know the percent of boot sales to total sales for each region. After some thought, we wondered if we could embed a conditional CASE-WHEN within our summary function. The summary function would create a new column which would contain only the boot sales for each region. Next we could divide that new column by the total sales to get our percentage of boot sales. Below is the PROC SQL query that was used to generate the data in figure 7.

proc sql outobs=5;
  select region,
      sum(sales) as sales format=dollar16,
      sum(case when Product='Boot' then sales else 0 end) as boot_sales format=dollar16,
      sum(case when Product='Boot' then sales else 0 end) / sum(sales) as boot_sales_ratio format=percent10.2
  from sashelp.shoes
  group by region;
quit;

Figure 7: CASE-WHEN with GROUP BY

<table>
<thead>
<tr>
<th>Region</th>
<th>sales</th>
<th>boot_sales</th>
<th>boot_sales_ratio</th>
</tr>
</thead>
<tbody>
<tr>
<td>Africa</td>
<td>$2,342,588</td>
<td>$119,835</td>
<td>5.12%</td>
</tr>
<tr>
<td>Asia</td>
<td>$460,231</td>
<td>$62,708</td>
<td>13.63%</td>
</tr>
<tr>
<td>Canada</td>
<td>$4,255,712</td>
<td>$385,613</td>
<td>9.06%</td>
</tr>
<tr>
<td>Central America/Caribbean</td>
<td>$3,857,753</td>
<td>$190,743</td>
<td>5.21%</td>
</tr>
<tr>
<td>Eastern Europe</td>
<td>$2,394,940</td>
<td>$306,705</td>
<td>12.81%</td>
</tr>
</tbody>
</table>

SQL IN DATABASE MANAGEMENT SYSTEMS

Much of this paper showed the reader how to use PROC SQL instead of other SAS procedures, to generate new tables and reports. One neglected benefit for using PROC SQL, is the ability to port the SQL code to 3rd party databases. As an analyst in our fictitious shoe company, our adhoc reports gathered visibility, and the management team wants the database system to automatically create out our reports. We can accomplish this by stripping out all
the SAS only SQL syntax. We remove the format and label options and now we have ANSI SQL which can be sent to the database team for implementation. ANSI SQL is the industry standard for SQL syntax and is generally accepted by most database vendors.

Also, most of the PROC SQL queries ran in a single step without creating temporary tables. Some database systems will not allow the user to create temporary tables without permission from the database administrator. Structuring a query that needs a temporary table get the desired output will put a hindrance to our analysis. By using in-line views instead of temporary tables for our queries, we can solve that potential problem before it arises.

CONCLUSION

The PROC SQL syntax is myriad and complex but with some simple techniques, we can leverage this powerful procedure to bend the data to our will. In this paper a few examples were shown to enable the reader to start using PROC SQL for common data processing tasks: summarizing data, finding duplicates, and merging data. The benefits of in-line views were shown not just to make our code elegant but also solving real world problem of porting our code to a third party database. By stripping out the SAS specific pieces of SQL, we can use the basic and advanced SQL queries to query any data from any ANSI SQL system, thus making us more well rounded analysts.

REFERENCES


CONTACT INFORMATION

Your comments and questions are valued and encouraged. Contact the author at:

Darryl Putnam
CACI Inc.
6835 Deerpath Road
Elkridge, MD 21075
Work Phone: 410-762-6535
E-mail: dputnam@cai.com

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