DSCI 325: Handout 16 – Introduction to PROC SQL

Spring 2017

SQL AND THE SQL PROCEDURE

SQL stands for Structured Query Language, which is a widely used language for retrieving, joining, and updating data stored in databases. PROC SQL is SAS's implementation of this widely used language. We have already discussed the use of DATA steps for reading and/or modifying data and the use of SAS PROCs to perform specific analyses or functions (e.g., sorting, printing, or writing reports). In this handout, we will discuss how PROC SQL can be used as an alternative to these other procedures.

It may be helpful for you to understand the following terminology before discussing PROC SQL in detail:

SQL Term	*SAS Term*
Table	SAS data set
Row	Observation
Column	Variable

USING PROC SQL

As stated above, PROC SQL is SAS's implementation of the Structured Query Language. This procedure takes on the following general form:

```
PROC SQL;
  sql-statement;
QUIT;
```

The *sql-statement* referenced above can be any SQL clause (e.g., ALTER, CREATE, DELETE, DESCRIBE, DROP, INSERT, SELECT, UPDATE, or VALIDATE). Consider the next example which implements the CREATE, INSERT, AND SELECT clauses.

Creating a Table from Scratch with PROC SQL

Suppose you want to create a simple SAS dataset containing two variables (*Name* and *Age*) and three observations.

```
Adelyn      3
Ava         1
Isaac       2
```

We could either use a SAS DATA step or PROC SQL:

```
DATA kids;
INPUT Name $ Age;
DATALINES;
Adelyn       3
Ava          1
Isaac        2
;

PROC PRINT data=kids;
TITLE 'The Kids';
RUN;
```

Output:

The Kids

Obs	Name	Age
1	Adelyn	3
2	Ava	1
3	Isaac	2

```
PROC SQL;
  CREATE TABLE kids
  (Name char, Age num);
  INSERT INTO kids
    VALUES ('Adelyn', 3)
      VALUES ('Ava', 1)
      VALUES ('Isaac', 2);

  TITLE 'The Kids';
  SELECT *
  FROM kids;
QUIT;
```

Output:

The Kids

Name	Age
Adelyn	3
Ava	1
Isaac	2

Creating a Table from an Existing Data Set

The following statements can be used to read in the *CarAccidents* data set (which already exists in my permanent library) and to save the results to a data set named *CarAccidents2*. The code on the left shows how this is accomplished with the DATA step, while the code on the right shows how this is done with PROC SQL.

```
DATA CarAccidents2;
  SET Hooks.CarAccidents;
PROC PRINT DATA = CarAccidents2;
RUN;
```

The first five observations are shown below:

Obs	ID	Gender	Seat_Belt	Age_Group	Cell_Phone_Involved	Cell_Phone_Usage
1	1	Female	Yes	16-18	No	None
2	2	Female	Yes	Over 30	No	None
3	3	Male	No	16-18	Yes	Text
4	4	Female	Yes	18-30	Yes	Talk
5	5	Female	Yes	18-30	No	None

```
PROC SQL;
CREATE TABLE CarAccidents2 AS
SELECT *
  FROM Hooks.CarAccidents;
QUIT;
```

Note that the above PROC SQL statements do not print any results to the Results Viewer window. Modify the code below to print the data set.

```
PROC SQL;
CREATE Table CarAccidents2 AS
SELECT *
  FROM Hooks.CarAccidents;




QUIT;
```

Keeping only Certain Columns of a Table

Next, suppose you wanted to include only the columns for Gender and Seat_Belt in the CarAccidents2 data set. This could be accomplished in either of the following ways:

```
DATA CarAccidents2;
  SET Hooks.CarAccidents
  (keep = Gender Seat_Belt);
PROC PRINT DATA = CarAccidents2;
RUN;
```

```
PROC SQL;
CREATE Table CarAccidents2 AS
SELECT Gender, Seat_Belt
  FROM Hooks.CarAccidents;
QUIT;
```

"Printing" Results With PROC SQL

The following example contrasts the PROC PRINT step with PROC SQL for printing data tables.

```
PROC PRINT DATA = Hooks.CarAccidents
LABEL;
LABEL Seat_Belt = Seat Belt Use;
LABEL Cell_Phone_Involved = Cell
Phone Involved?;
VAR Gender Seat_Belt
Cell_Phone_Involved;
RUN;
```

Obs	Gender	Seat Belt Use	Cell Phone Involved?
1	Female	Yes	No
2	Female	Yes	No
3	Male	No	Yes
4	Female	Yes	Yes
5	Female	Yes	No

```
PROC SQL;
SELECT Gender,
   Seat_Belt LABEL = 'Seat Belt Use',
   Cell_Phone_Involved LABEL = 'Cell
   Phone Involved?'
FROM Hooks.CarAccidents;
QUIT;
```

Gender	Seat Belt Use	Cell Phone Involved?
Female	Yes	No
Female	Yes	No
Male	No	Yes
Female	Yes	Yes
Female	Yes	No

Subsetting a Table

Notice that in all of the above programs involving PROC SQL, the SELECT statement must contain both a

- SELECT clause, which lists the names of the column(s) of interest
- FROM clause, which lists the table in which the column(s) reside(s).

You can also include a WHERE statement to restrict the data that you retrieve. For example, suppose that you wanted to read in only the females from the CarAccidents Table. You could accomplish this with either the DATA step (as seen in earlier handouts) or with PROC SQL:

Option 1 with DATA step:

```
DATA CarAccidents2;
  SET Hooks.CarAccidents;
  WHERE Gender = 'Female';
PROC PRINT DATA = CarAccidents2;
RUN;
```

Option 2 with DATA step:

```
DATA CarAccidents2;
  SET Hooks.CarAccidents;
  IF Gender = 'Female';
PROC PRINT DATA = CarAccidents2;
RUN;
```

```
PROC SQL;
CREATE Table CarAccidents2 AS
SELECT *
  FROM Hooks.CarAccidents
  WHERE Gender = 'Female';
QUIT;
```

Comment: Note that although the results are the same for this example, there are some differences in how the WHERE statement and the IF statement work in the DATA step:

- The subsetting IF can appear only in a DATA step, whereas the WHERE statement can be used in DATA steps or other PROC steps.

- The WHERE statement is more efficient because it avoids reading unwanted rows.

- The WHERE statement can select rows only from existing SAS tables. The IF statement, however, can be used to select rows from existing tables or from raw data files being read in with INPUT statements.

Sorting Data with PROC SQL

PROC SQL can be used to achieve the same result as PROC SORT. For example, consider the following programs:

```
PROC SORT DATA = Hooks.CarAccidents;
  BY Age_Group Gender;
PROC PRINT DATA = Hooks.CarAccidents;
RUN;
```

```
PROC SQL;
  SELECT *
  FROM Hooks.CarAccidents
  ORDER BY Age_Group, Gender;
QUIT;
```

Creating New Columns in a Table

PROC SQL can also be used to calculate new columns by assigning an expression to an item name. For example, consider the following programs:

```
DATA Grades2;
  SET Hooks.Grades;
  TotalExam =
  SUM(Exam1,Exam2,Exam3,Final);
RUN;
```

```
PROC SQL;
CREATE Table Grades2 AS
  SELECT * ,
  Exam1 + Exam2 + Exam3 + Final AS
  TotalExam
  FROM Hooks.Grades;
QUIT;
```

Creating New Columns in a Table with Conditional Logic

The CASE expression in SQL assigns values to fields in the same way an IF-THEN-ELSE statement works in a DATA step. Consider the following programs.

```
DATA Final;
  SET Hooks.Grades (keep=Final);
  IF Final >= 90 then FinalExamGrade = 'A';
  ELSE IF Final >= 80 then FinalExamGrade='B';
  ELSE IF Final >= 70 then FinalExamGrade='C';
  ELSE IF Final >= 60 then FinalExamGrade='D';
  ELSE FinalExamGrade = 'F';
RUN;
```

```
PROC SQL;
  CREATE TABLE Final AS
  SELECT Final, Case
    WHEN Final >= 90 then 'A'
    WHEN Final >=80 and Final < 90 then 'B'
    WHEN Final >=70 and Final < 80 then 'C'
    WHEN Final >=60 and Final < 70 then 'D'
    ELSE 'F'
  END AS FinalExamGrade
FROM Hooks.Grades;
QUIT;
```

Calculating Summary Statistics

Recall that by default the MEANS procedure will produce N (counts), the mean, the standard deviation, the min, and the max for all numeric variables in a data set. We can use the VAR statement in PROC MEANS to specify that these quantities be calculated for only certain variables (e.g., for only the *Final* variable, as shown below). Note that these quantities can also be calculated using PROC SQL.

```
PROC MEANS DATA = Hooks.Grades;
  VAR Final;
RUN;
```

The MEANS Procedure

Analysis Variable : Final				
N	Mean	Std Dev	Minimum	Maximum
134	69.1791045	20.3351064	0	100.0000000

```
PROC SQL;
SELECT count(Final) as N,
       avg(Final) as Mean,
       std(Final) label 'Std Dev',
       min(Final) as Min,
       max(Final) as Max
FROM Hooks.Grades;
QUIT;
```

N	Mean	Std Dev	Min	Max
134	69.1791	20.33511	0	100

Question: Add the following to the above PROC SQL code. How does this change the output?

```
PROC SQL;
SELECT 'Final' as Variable,
         count(Final) as N,
         avg(Final) as Mean,
         std(Final) label 'Std Dev',
         min(Final) as Min,
         max(Final) as Max
FROM Hooks.Grades;
QUIT;
```

Calculating Summary Statistics by Group

Recall that we have used the BY statement in PROC MEANS. We can obtain similar results with the GROUP BY clause in PROC SQL. Consider the following programs and output.

```
PROC SORT DATA = Hooks.NYC_Trees;
  BY Condition;
PROC MEANS DATA = Hooks.NYC_Trees N
  MEAN STD;
  VAR FoliageDensity;
  BY Condition;
RUN;
```

The MEANS Procedure

Condition=Excellent

Analysis Variable : FoliageDensity		
N	Mean	Std Dev
34	82.3529412	14.2081546

Condition=Good

Analysis Variable : FoliageDensity		
N	Mean	Std Dev
254	56.5748031	18.4337376

Condition=Poor

Analysis Variable : FoliageDensity		
N	Mean	Std Dev
31	32.0967742	13.3400521

```
PROC SQL;
SELECT Condition,
       count(FoliageDensity) AS N,
       avg(FoliageDensity) AS Mean,
       std(FoliageDensity) AS std
       LABEL = 'Std Dev'
FROM Hooks.NYC_Trees
GROUP BY Condition ;
QUIT;
```

Condition	N	Mean	Std Dev
Excellent	34	82.35294	14.20815
Good	254	56.5748	18.43374
Poor	31	32.09677	13.34005

Selecting Unique Values of One or More Columns

You can use the DISTINCT clause to list only unique values of a variable. For example, consider the following programming statements and their corresponding output:

```
PROC SQL;
SELECT DISTINCT Condition
  FROM Hooks.NYC_Trees;
QUIT;
```

Condition
Excellent
Good
Poor

```
PROC SQL;
SELECT DISTINCT Condition, Native
  FROM Hooks.NYC_Trees;
QUIT;
```

Condition	Native
Excellent	NO
Excellent	YES
Good	NO
Good	YES
Poor	NO
Poor	YES

Compare the above results to the following:

```
PROC SQL;
SELECT Condition, Native, Count(*)
  FROM Hooks.NYC_Trees
  GROUP BY Condition, Native;
QUIT;
```

Condition	Native	
Excellent	NO	20
Excellent	YES	14
Good	NO	205
Good	YES	49
Poor	NO	21
Poor	YES	10

Concatenating Tables with PROC SQL

Recall the following example from Handout 12:

Emps

Obs	FirstName	Gender	HireYear
1	Matt	M	2000
2	Melia	F	2011
3	Josh	M	2011
4	Corey	M	2011

Emps2013

Obs	FirstName	Gender	HireYear
1	Kristi	F	2012
2	Carrie	F	2012

These data sets can be concatenated using either the DATA step (as was done earlier in the semester) or using PROC SQL:

```
DATA EmpsAll;
SET Emps Emps2013;
RUN;
```

```
PROC SQL ;
CREATE TABLE EmpsAll AS
SELECT * FROM Hooks.Emps
UNION
SELECT * FROM Hooks.Emps2013
QUIT;
```

The result is shown below:

EmpsAll

Obs	FirstName	Gender	HireYear
1	Carrie	F	2012
2	Corey	M	2011
3	Josh	M	2011
4	Kristi	F	2012
5	Matt	M	2000
6	Melia	F	2011

Merging Tables with PROC SQL

Once again, consider the following data sets from Handout 12:

EmpsAU

Obs	first	Gender	EmpID
1	Togar	M	121150
2	Kylie	F	121151
3	Birin	M	121152

PhoneC

Obs	Phone	EmpID
1	+61(2)5555-1793	121150
2	+61(2)5555-1849	121151
3	+61(2)5555-1348	121153

We can merge these two data sets using either the MERGE statement in a DATA step or by using an **INNER JOIN** in PROC SQL:

```
DATA EmpsAUC;
  MERGE Hooks.EmpsAU (IN = a)
        Hooks.PhoneC (IN = b);
  BY EmpID;
  IF a = 1 and b = 1;
RUN;
```

Option 1

```
PROC SQL;
CREATE TABLE EmpsAUC AS
SELECT *
FROM Hooks.EmpsAU, Hooks.PhoneC
WHERE EmpsAU.EmpID=PhoneC.EmpID;
QUIT;
```

Option 2

```
PROC SQL;
CREATE TABLE EmpsAUC AS
SELECT * FROM Hooks.EmpsAU
INNER JOIN Hooks.PhoneC
ON EmpsAU.EmpID = PhoneC.EmpID;
QUIT;
```

Output:

Obs	first	Gender	EmpID	Phone
1	Togar	M	121150	+61(2)5555-1793
2	Kylie	F	121151	+61(2)5555-1849

The previous example showed what is known as an INNER JOIN (i.e., the final table contains only the rows that match in both of the original tables). We can also use either the DATA step or PROC SQL to accomplish a **FULL OUTER JOIN**:

```
DATA EmpsAUC;
MERGE Hooks.EmpsAU Hooks.PhoneC;
BY EmpID;
RUN;
```

```
PROC SQL;
CREATE TABLE EmpsAUC AS
SELECT * FROM Hooks.EmpsAU
FULL OUTER JOIN Hooks.PhoneC
ON EmpsAU.EmpID = PhoneC.EmpID;
QUIT;
```

Output:

Obs	first	Gender	EmpID	Phone
1	Togar	M	121150	+61(2)5555-1793
2	Kylie	F	121151	+61(2)5555-1849
3	Birin	M	121152	
4			121153	+61(2)5555-1348

Note that this also works for a one-to-many merge. Recall the following example.

EmpsAU

Obs	First	Gender	EmpID
1	Togar	M	121150
2	Kylie	F	121151
3	Birin	M	121152

PhoneHW

Obs	EmpID	Type	Phone
1	121150	Home	+61(2)5555-1793
2	121150	Work	+61(2)5555-1794
3	121151	Home	+61(2)5555-1849
4	121151	Work	+61(2)5555-1850
5	121152	Home	+61(2)5555-1665
6	121152	Work	+61(2)5555-1666

Consider the following programming statements.

```
DATA EmpsAUHW;
MERGE EmpsAU PhoneHW;
BY EmpID;
RUN;
```

```
PROC SQL;
CREATE TABLE EmpsAUC AS
SELECT * FROM Hooks.EmpsAU
FULL OUTER JOIN Hooks.PhoneHW
ON EmpsAU.EmpID = PhoneHW.EmpID;
QUIT;
```

Both produce the following output:

Obs	First	Gender	EmpID	Type	Phone
1	Togar	M	121150	Home	+61(2)5555-1793
2	Togar	M	121150	Work	+61(2)5555-1794
3	Kylie	F	121151	Home	+61(2)5555-1849
4	Kylie	F	121151	Work	+61(2)5555-1850
5	Birin	M	121152	Home	+61(2)5555-1665
6	Birin	M	121152	Work	+61(2)5555-1666

Using PROC SQL for Subqueries

Suppose the following three data sets exist in a permanent SAS library.

- MajorCurrSat
- GraduateSucc
- StudentInfo

Consider the following code, which uses a subquery.

```
PROC SQL;
 SELECT  MCS_Semester, Avg(MCS_Q1), Avg(MCS_Q2), Avg(MCS_Q3)
   FROM Hooks.MajorCurrSat
   WHERE Student_ID in
         (select Student_ID from Hooks.GraduateSucc WHERE
          GS_Employed='Yes')
   GROUP BY MCS_Semester;
RUN;
```

Tasks:

1. Run the above code and verify the following result:

MCS_Semester			
20025	2.954545	3.181818	3.227273
20035	3.088889	2.822222	3.022222
20045	3.09375	3.052083	3.115789
20055	3.090909	3.136364	3.245455
20065	3.232143	3.142857	3.25
20075	3.317757	3.242991	3.28972
20085	3.433333	3.244444	3.3
20095	3.132075	3.113208	3.169811

2. How would you modify the code in order to get column headers to appear?

3. Run the code without the subquery, which is highlighted in yellow. You should get the following result. What is the difference between this result and that obtained in Task 1?

MCS_Semester			
20025	2.8	3.16	3.2
20035	3	2.878788	3.060606
20045	3.05042	3.092437	3.110169
20055	3.020979	3.118881	3.20979
20065	3.23741	3.172662	3.294964
20075	3.268	3.224	3.28
20085	3.398305	3.254237	3.305085
20095	3.185714	3.142857	3.242857

Finally, note that we could also use a subquery to create a subset and put it into its own table. This is shown in the next example.

```
PROC SQL;
 CREATE TABLE Hooks.MajorCurrSatYes
   LIKE Hooks.MajorCurrSat;

   DESCRIBE table Hooks.MajorCurrSatYes;

QUIT;


PROC SQL;
 TITLE 'Inserting Rows into a Table';
 INSERT into Hooks.MajorCurrSatYes
 SELECT  *
   FROM Hooks.MajorCurrSat
   WHERE Student_ID in
        (select Student_ID from Hooks.GraduateSucc WHERE
          GS_Employed='Yes');
QUIT;
```

Some Practice Problems:

1. Note that we could use PROC MEANS as shown below to compute the average score for Questions 1, 2, and 3 from the MajorCurr Sat survey by Semester. Write a PROC SQL program that also accomplishes this task.

```
PROC SORT DATA=Hooks.MajorCurrSat OUT=Hooks.MajorCurrSat2;
 BY MCS_Semester;
RUN;

PROC MEANS DATA=Hooks.MajorCurrSat2;
 BY MCS_Semester;
 VAR MCS_Q1 MCS_Q2 MCS_Q3;
 OUTPUT OUT=Mean_Output
 MEAN(MCS_Q1 MCS_Q2 MCS_Q3) = AvgMCS_Q1 AvgMCS_Q2 AvgMCS_Q3;
RUN;

PROC PRINT DATA=Mean_Output;
 VAR MCS_Semester AvgMCS_Q1 AvgMCS_Q2 AvgMCS_Q3;
RUN;
```

2. Note that PROC FREQ could be used to analyze the distribution of GS_EmploymentRelated, as shown below.

```
PROC FREQ DATA=Hooks.GraduateSucc;
TABLE GS_EmploymentRelated;
RUN;
```

GS_EmploymentRelated	Frequency	Percent	Cumulative Frequency	Cumulative Percent
Related	471	75.48	471	75.48
Somewhat Related	74	11.86	545	87.34
Unrelated	79	12.66	624	100.00

Write a PROC SQL program that calculates the frequency (i.e., count) of each category of GS_EmploymentRelated.