SAS® GLOBAL FORUM 2020 #SASGF

Paper SAS 5167-2020

Step-by-Step SQL Procedure

Charu Shankar, SAS Institute Inc.

ABSTRACT

PROC SQL is a powerful query language that can sort, summarize, subset, join, and print results all in one step. Users who are continuously improving their analytical processing will benefit from this hands-on workshop. In this paper, participants learn the following elements to master PROC SQL:

1. Understand the syntax order in which to submit queries to PROC SQL
2. Summarize data using Boolean operations
3. Manage metadata using dictionary tables
4. Join tables using join conditions like inner join and reflexive join
5. Internalize the logical order in which PROC SQL processes queries

INTRODUCTION

PROC SQL is the language of databases. After teaching at SAS for more than 10 years to thousands of learners, this instructor has collected many best practices from helping customers with real-world business problems. This paper illustrates practices such as how to make coding life easy with mnemonics to recall the order of statements in SQL, and how to leverage simple yet elegant techniques such as Boolean logic in SQL. Data used in this paper can be downloaded from this Github Repository: https://github.com/CharuSAS/SQL.

UNDERSTAND THE SYNTAX ORDER IN WHICH TO SUBMIT QUERIES TO PROC SQL

Every computer language has syntax order that is uniquely its own. Trying to remember the syntax is sometimes not easy for beginners and even those fluent in multiple languages, human or computer. For some help in memory recall, try my mnemonic to remember the syntax order of SQL.

SO	**SELECT** *object-item* <, ...*object-item*>
FEW	**FROM** *from-list*
WORKERS	<**WHERE** *sql-expression*>
GO	<**GROUP BY** *object-item* <, … *object-item* >>
HOME	<**HAVING** *sql-expression*>
ON TIME	<**ORDER BY** *order-by-item* <*DESC*> <, …*order-by-item*>>**;**

Figure 1: PROC SQL Mnemonic

Here is a PROC SQL query in its entirety. SELECT and FROM are mandatory statements in any SQL query. Anything in triangular brackets is optional.

```
PROC SQL;
    SELECT object-item <, ...object-item>
        FROM from-list
            <WHERE sql-expression>
                <GROUP BY object-item <, … object-item >>
                    <HAVING sql-expression>
                        <ORDER BY order-by-item <DESC>
                                <, …order-by-item>>;
```

Figure 2: PROC SQL Syntax Order

A *SELECT statement* is used to query one or more tables.

The FROM clause specifies the tables that are required for the query.

The WHERE clause specifies data that meets certain conditions.

The GROUP BY clause groups data for processing.

The HAVING clause specifies groups that meet certain conditions.

The ORDER BY clause specifies an order for the data.

SUMMARIZE DATA USING BOOLEAN OPERATIONS

Hands down, summarizing data using the Boolean gate in PROC SQL has to be my all-time favorite technique. When I fell in love with its elegance, I captioned my blog captioned "No. 1 Best programming technique for 2012." It was easily my #1 best technique for life, but I thought I would keep myself open to new learning! Read on to learn more about this magic.

Summarizing Data

The Boolean is simply the digital computing world's way of converting everything to 0s and 1s. A yes is a one, and a no is a zero.

Grouping Data

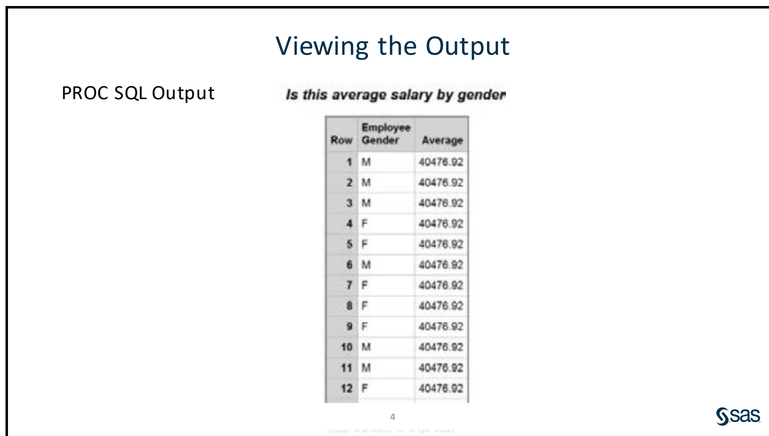
Let's begin with a simple business scenario to understand grouping first. We have been asked to produce a report that determines the average salary by gender.

How many rows does this query create?

```
title ' Is this average salary by gender';
proc sql number;
   select Employee_Gender, avg(Salary) as Average
      from SGF2020.employee_information
         where Employee_Term_Date is missing;
quit;
```

Display 1: Code for Average Salary by Gender

The result is not quite as expected. Instead of receiving 2 rows of data, the output contains 308 rows. This is the number of rows in the SGF2020.employee_information table. Also, the average is not an average for each gender, rather the average for the entire table.



Row	Employee Gender	Average
1	M	40476.92
2	M	40476.92
3	M	40476.92
4	F	40476.92
5	F	40476.92
6	M	40476.92
7	F	40476.92
8	F	40476.92
9	F	40476.92
10	M	40476.92
11	M	40476.92
12	F	40476.92

Figure 3: Unexpected Output for Average Salary by Gender

The GROUP BY Clause

You can use the GROUP BY clause to do the following:

- classify the data into groups based on the values of one or more columns
- calculate statistics for each unique value of the grouping columns

```
title "Average Salary by Gender";
proc sql;
   select Employee_Gender as Gender, avg(Salary) as Average
   from SGF2020.employee_information
      where Employee_Term_Date is missing
         group by Employee_Gender;
quit;
```

Display 2: Correct Code for Average Salary by Gender

The results are more satisfactory this time, with two rows of data.

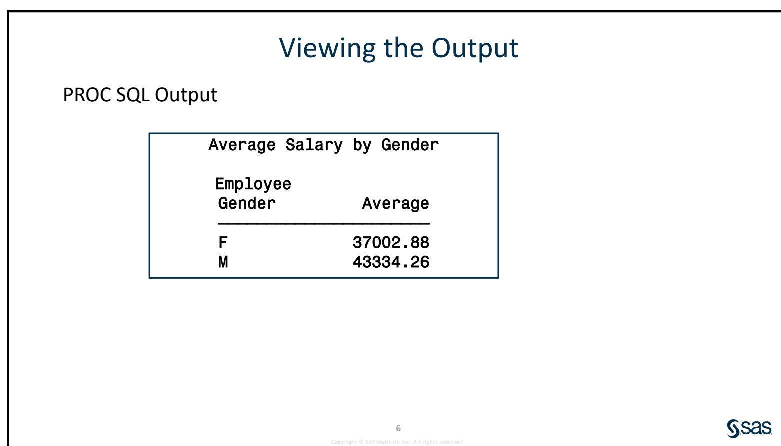


Figure 4: Correct Output for Average Salary by Gender

Let’s move on to the next level of complexity. We have been tasked to produce a report showing the count of employees in departments that have at least 25 people. Display the results in descending order by count.

A first step would be to count the number of employees for each department.

```
title 'Employee count by department';
proc sql;
   select Department, count(*) as Count
      from SGF2020.employee_information
         group by Department;
quit;
```

Display 3: Code for Employee Counts by Department

Viewing the Output

PROC SQL Output

Employee count by department

Department	Count
Accounts	17
Accounts Management	9
Administration	34
Concession Management	11
Engineering	9
Executives	4
Group Financials	3
Group HR Management	18
IS	25
Logistics Management	14
Marketing	20
Purchasing	18
Sales	201
Sales Management	11
Secretary of the Board	2
Stock & Shipping	26
Strategy	2

Figure 5: Employee Counts by Department

In the next step, we control the result to include only the departments that have at least 25 people, with the departments in decreasing order. To do this, we use the *HAVING clause*, which subsets groups based on the expression value.

```
title 'Employee counts by department in departments with at least 25
employees';
proc sql;
   select Department, count(*) as Count
      from SGF2020.employee_information
         group by Department
            having Count ge 25
               order by Count desc;
quit;
```

Display 4: Code for Employee Counts by Department with at Least 25 Employees

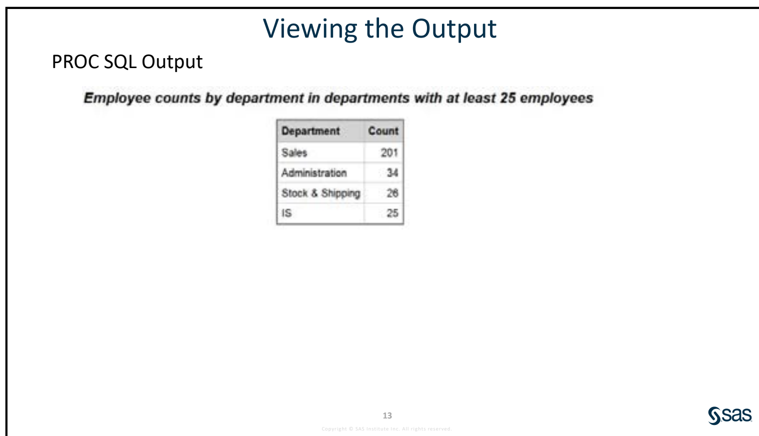


Figure 6: Employee Counts by Department with at Least 25 Employees

Have you ever been challenged with a business scenario where you had to subset data to return both the haves and the have nots?

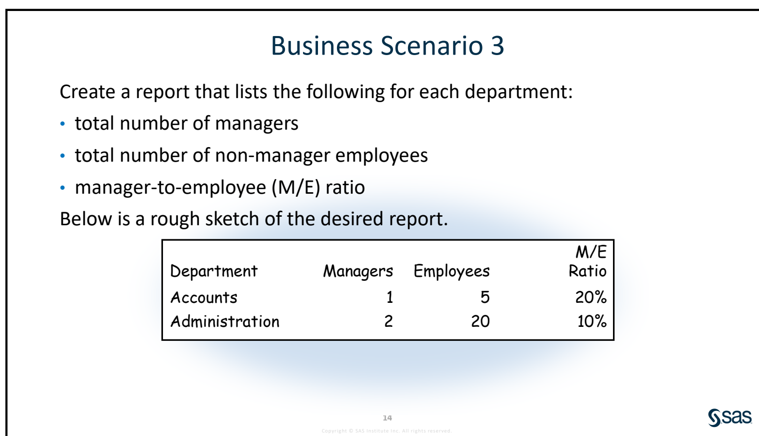


Figure 7: Business Scenario for Total Number of Managers and Employees

How will you go about extracting both the managers and the employees and stick them all on the same line?

First, we use the FIND function to find all managers.

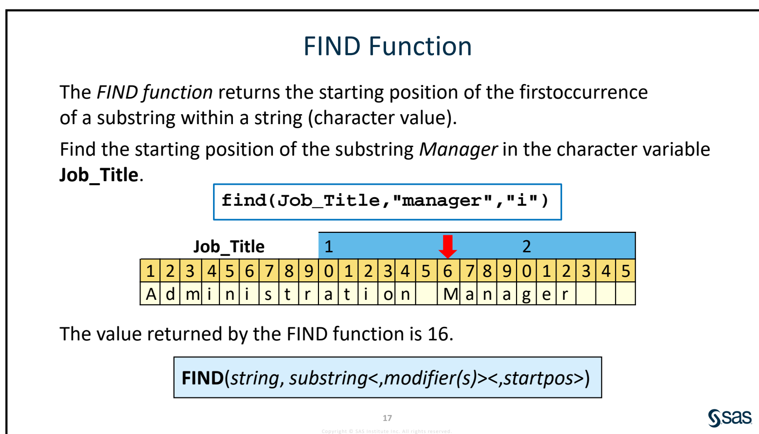


Figure 8: FIND Function

Here is the classic Boolean put to good use to determine whether an employee is a manager. If Job_Title contains *Manager*, the value is 1. If it doesn't contain *Manager*, the value is 0.

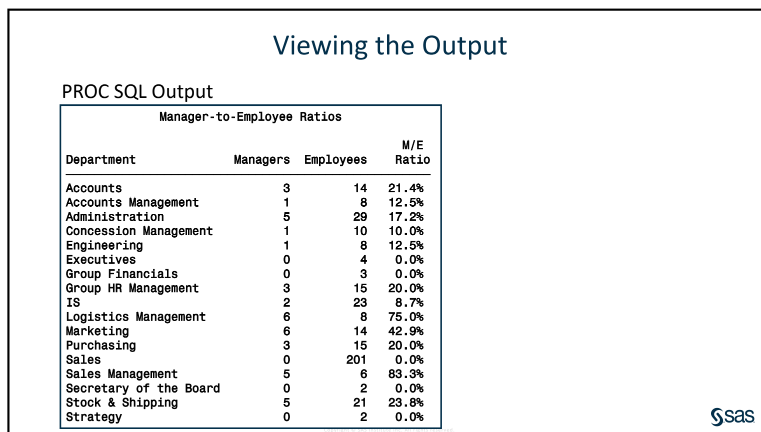
```
title 'Manager or not';
proc sql;
   select Department, Job_Title,
      (find(Job_Title,"manager","i")>0) "Manager"
         from SGF2020.employee_information;
quit;
```

Display 5: Code to Write a Boolean Expression

Now simply calculate the statistics by wrapping the Boolean expressions with the SUM function.

```
title "Manager-to-Employee Ratios";
proc sql;
   select Department,
      sum((find(Job_Title,"manager","i")>0))as Managers,
      sum((find(Job_Title,"manager","i")=0))as Employees,
      calculated Managers/calculated Employees
      "M/E Ratio" format=percent8.1
         from SGF2020.employee_information
         group by Department;
quit;
```

Display 6: Code to Summarize Data Using the Boolean



Manager-to-Employee Ratios

Department	Managers	Employees	M/E Ratio
Accounts	3	14	21.4%
Accounts Management	1	8	12.5%
Administration	5	29	17.2%
Concession Management	1	10	10.0%
Engineering	1	8	12.5%
Executives	0	4	0.0%
Group Financials	0	3	0.0%
Group HR Management	3	15	20.0%
IS	2	23	8.7%
Logistics Management	6	8	75.0%
Marketing	6	14	42.9%
Purchasing	3	15	20.0%
Sales	0	201	0.0%
Sales Management	5	6	83.3%
Secretary of the Board	0	2	0.0%
Stock & Shipping	5	21	23.8%
Strategy	0	2	0.0%

Figure 9: Output Using Boolean Operations

This was just one way to use the Boolean. The expressions can be as complex as necessary.

MANAGE METADATA USING DICTIONARY TABLES

There is no magic pill that will forgive us for not knowing our data. "Know thy data" must be the most fundamental principle that cannot be ignored. In fact, I am going to go out on a limb here and say that this is the only rule that data workers must know. Everything else is SAS!

To help navigate through the inherited – and sometimes messy – data, my go-to suggestion is DICTIONARY tables. With the amount of heavy-duty metadata scouring that data workers perform, this is one tip you must see. I love DICTIONARY tables and cannot imagine life without them. When you see this confession revealed, I'm positive you will also feel the same way.

DICTIONARY tables are Read-Only metadata views that contain session metadata, such as information about SAS libraries, data sets, and external files in use or available in the current SAS session.

DICTIONARY tables are

- created at SAS session initialization
- updated automatically by SAS
- limited to Read-Only access.

You can query DICTIONARY tables with PROC SQL.

There can be more than 30 DICTIONARY tables. We will focus on two of the tables.

- DICTIONARY.TABLES - detailed information about tables
- DICTIONARY.COLUMNS - detailed information about all columns in all tables

To get to know the columns and what they stand for, query the DICTIONARY table first using the following code.

```
proc sql;
   describe table dictionary.tables;
quit;
```

Display 7: Code to Describe DICTIONARY Tables

Log

```
NOTE: SQL table DICTIONARY.TABLES was created like:
create table DICTIONARY.TABLES
  (libname char(8) label='Library Name',
   memname char(32) label='Member Name',
   ...
   crdate num format=DATETIME informat=DATETIME label='Date Created',
   modate num format=DATETIME informat=DATETIME label='Date Modified',
   nobs num label='Number of Physical Observations',
   obslen num label='Observation Length',
   nvar num label='Number of Variables', ...);
```

Display 8: Log to Describe DICTIONARY Tables

Let’s begin to understand the dictionary tables by querying all tables with an ID column.

```
title 'Tables Containing an ID Column';
proc sql;
   select memname 'Table Names', name
      from dictionary.columns
         where libname='SASHELP' and
            upcase(name) contains 'ID';
quit;
```

Display 9: Code to Query All Tables Containing an ID Column

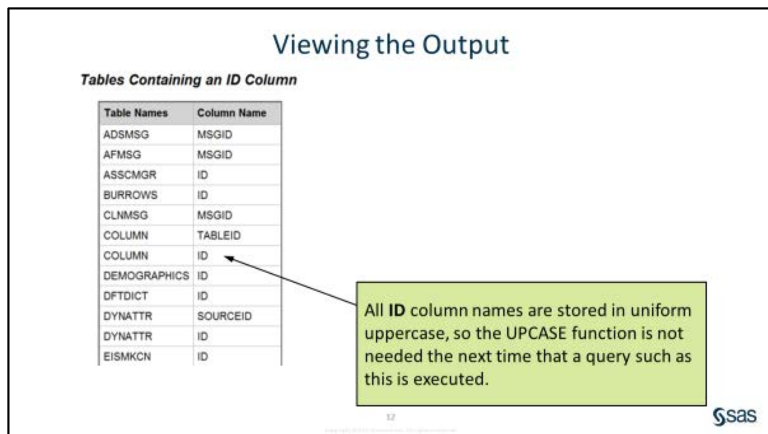


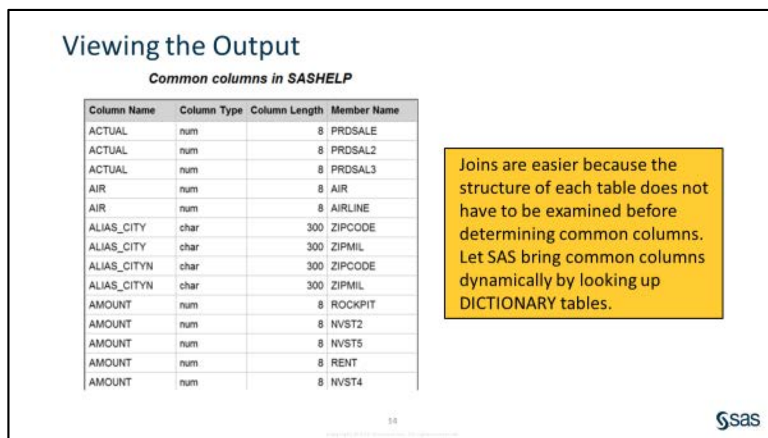
Table Names	Column Name
ADSMSG	MSGID
AFMSG	MSGID
ASSCMGR	ID
BURROWS	ID
CLNMSG	MSGID
COLUMN	TABLEID
COLUMN	ID
DEMOGRAPHICS	ID
DFTDICT	ID
DYNATTR	SOURCEID
DYNATTR	ID
EISMKCN	ID

Figure 10: PROC SQL Output Tables Containing an ID Column

However, you might have observed that this is something that PROC CONTENTS can do. It’s not something that impresses us as a niche value that DICTIONARY tables can add. Also, these past techniques work when you know the names of columns. What happens if you don’t know your data, and you want SAS to retrieve all same-named columns in a library. The real power of DICTIONARY tables reveals itself when we eliminate any manual work.

```
title 'Common columns in SASHELP';
proc sql;
select name, type, length, memname
   from dictionary.columns
      where libname='SASHELP’
         group by name
            having count(name) > 1;
quit;
```

Display 10: Code to Find Common Column Names Dynamically



Column Name	Column Type	Column Length	Member Name
ACTUAL	num	8	PRDSALE
ACTUAL	num	8	PRDSAL2
ACTUAL	num	8	PRDSAL3
AIR	num	8	AIR
AIR	num	8	AIRLINE
ALIAS_CITY	char	300	ZIPCODE
ALIAS_CITY	char	300	ZIPMIL
ALIAS_CITYN	char	300	ZIPCODE
ALIAS_CITYN	char	300	ZIPMIL
AMOUNT	num	8	ROCKPIT
AMOUNT	num	8	NVST2
AMOUNT	num	8	NVST5
AMOUNT	num	8	RENT
AMOUNT	num	8	NVST4

Figure 11: Common Column Names of Tables in the Sashelp Library

JOIN TABLES USING JOIN CONDITIONS LIKE INNER JOIN AND REFLEXIVE JOIN

SQL uses *joins* to combine tables horizontally. Requesting a join involves matching data from one row in one table with a corresponding row in a second table. Matching is typically performed on one or more columns in the two tables.

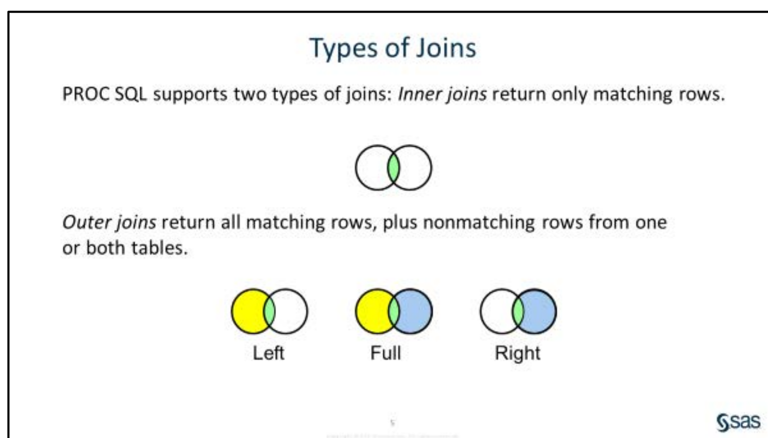


Figure 12: Inner and Outer Joins

Cartesian Product

A query that lists multiple tables in the FROM clause without a WHERE clause produces all possible combinations of rows from all tables. This result is called a *Cartesian product*.

```
title 'Combining data from multiple tables’;
proc sql;
   select *
      from SGF2020.customers, SGF2020.transactions;
quit;
```

Display 11: Code to Combine Data from Multiple Tables

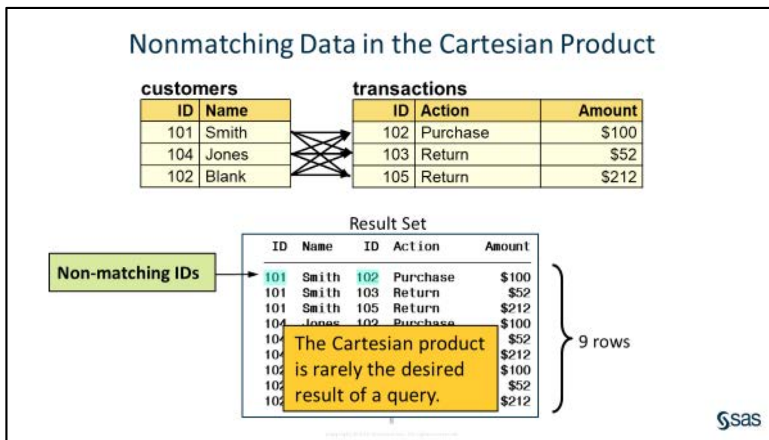


Figure 13: Cartesian Product

Inner Join

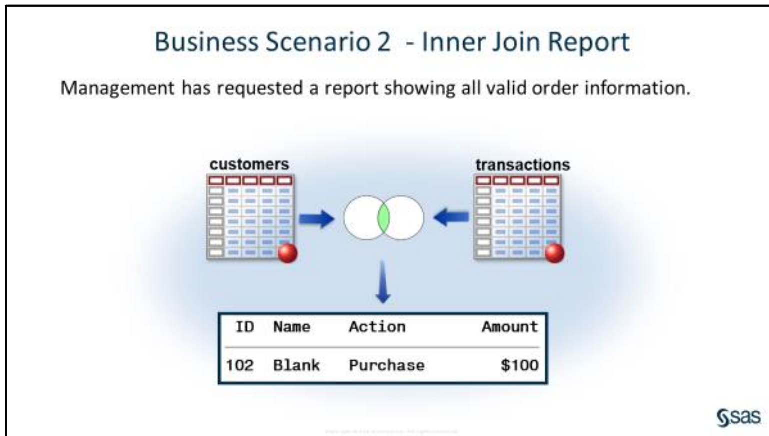


Figure 14: Inner Join Report

```
title 'Inner Join';
proc sql;
   select *
      from SGF2020.customers, SGF2020.transactions
         where customers.ID=
            transactions.ID;
quit;
```

Display 12: Code to craft inner join

While specifying same-named columns from more than one table, qualify the column name.

Completed Code

To display the ID column only once in the results, qualify the ID column in the SELECT clause.

customers

ID	Name
101	Smith
104	Jones
102	Blank

transactions

ID	Action	Amount
102	Purchase	$100
103	Return	$52
105	Return	$212

```
Title 'Qualifying column name in the SELECT';
proc sql;
  select customers.ID, Name, Action, Amount
    from SGF2020.customers, SGF2020.transactions
      where customers.ID=transactions.ID;
quit;
```

PROC SQL Output

ID	Name	Action	Amount
102	Blank	Purchase	$100

s104e03

Figure 15: Qualifying the ID Column in the SELECT Clause

Reflexive Joins

A *reflexive* join (also known as a *self-join*) is the joining of a table to itself.

The chief sales officer wants to have a report with the name of all sales employees and the name of each employee's direct manager.

Business Data

To return the employee name and the manager name, you need to read the **addresses** table twice.

1. Return the employee's ID and name.

addresses

EMP_ID	EMP_NAME
100	John
101	Sue

organization

EMP_ID	MGR_ID
100	101
101	57

EMP_ID	EMP_Name	MGR_ID	MGR_Name
100	John		

Figure 16: Return the E**mployee's ID and** Name

Business Data

To return the employee name and the manager name, you need to read the **addresses** table twice.

1. Return the employee's ID and name.
2. Determine the ID of the employee's manager.

addresses

EMP_ID	EMP_NAME
100	John
101	Sue

organization

EMP_ID	MGR_ID
100	101
101	57

EMP_ID	EMP_Name	MGR_ID	MGR_Name
100	John	101	

Figure 17: Determine the ID of the E**mployee's** Manager

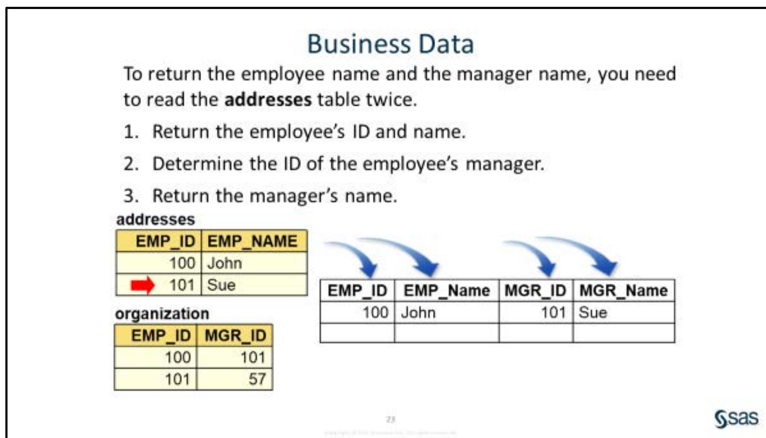
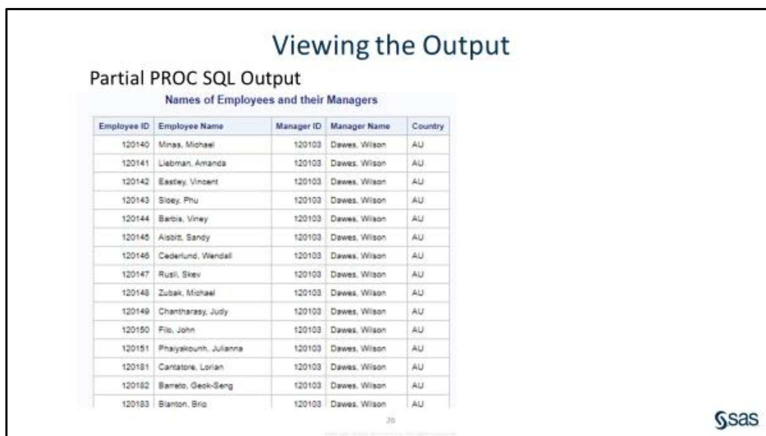


Figure 18**: Return the Manager's Name**

In order to read from the same table twice, it must be listed in the FROM clause twice. Here, a different table alias is required to distinguish the different uses.

```
proc sql;
   select e.Employee_ID "Employee ID",
          e.Employee_Name "Employee Name",
          m.Employee_ID "Manager ID",
          m.Employee_Name "Manager Name",
          e.Country
   from  SGF2020.employee_addresses as e,
         SGF2020.employee_addresses as m,
         SGF2020.employee_organization as o
   where e.Employee_ID=o.Employee_ID and
         o.Manager_ID=m.Employee_ID and
         Department contains 'Sales'
   order by Country,4,1;
quit;
```

Display 13: Code for Self-Join Using Different Table Aliases for The Same Table



Viewing the Output

Partial PROC SQL Output

Names of Employees and their Managers

Employee ID	Employee Name	Manager ID	Manager Name	Country
120140	Minas, Michael	120103	Dawes, Wilson	AU
120141	Liebman, Amanda	120103	Dawes, Wilson	AU
120142	Eastley, Vincent	120103	Dawes, Wilson	AU
120143	Sloey, Phu	120103	Dawes, Wilson	AU
120144	Barbis, Viney	120103	Dawes, Wilson	AU
120145	Aisbitt, Sandy	120103	Dawes, Wilson	AU
120146	Cederlund, Wendall	120103	Dawes, Wilson	AU
120147	Rusli, Skev	120103	Dawes, Wilson	AU
120148	Zubak, Michael	120103	Dawes, Wilson	AU
120149	Chantharasy, Judy	120103	Dawes, Wilson	AU
120150	Filo, John	120103	Dawes, Wilson	AU
120151	Phaiyakounh, Julianna	120103	Dawes, Wilson	AU
120181	Cantatore, Lorian	120103	Dawes, Wilson	AU
120182	Barreto, Geok-Seng	120103	Dawes, Wilson	AU
120183	Blanton, Brig	120103	Dawes, Wilson	AU

Figure 19: Self-Join Output

INTERNALIZE THE PROC SQL LOGICAL QUERY PROCESSING ORDER

In an earlier section, we discussed PROC SQL's syntax order. But the logical query processing order, which is the conceptual interpretation order, is as follows:

```
5  SELECT
1     FROM
2        WHERE
3           GROUP BY
4              HAVING
6                 ORDER BY
```

Display 14: PROC SQL Logical Query Processing Order

Thinking Like SQL – Logical Query Processing Order

Each phase operates on one or more tables as inputs and returns a virtual table as output. The output table of one phase is considered the input to the next phase.
Consider the following query as an example.

```
proc sql;
   SELECT country,YEAR(emphiredate)AS yearhired,COUNT(*)AS numemp
      FROM SGF2020.logicalq
         WHERE emphiredate >= "1jan2009"d
            GROUP BY country, yearhired
               HAVING COUNT(*) > 1
                  ORDER BY country , yearhired DESC;
QUIT;
```

Display 15: Example Code

1. Evaluate the FROM Clause

In the first phase, the FROM clause is evaluated. Indicate the tables to query and table operators like joins if applicable.

The output of this phase is a table result with all rows from the input table.

That's the case in the following query: the input is the SGF2020.country (322 rows), and the output is a table result with all 322 rows (only a subset of the attributes are shown).

```
proc sql;
create table sgf2020.logicalq as
  SELECT empid, country,emphiredate
    FROM SGF2020.country;
quit;
```

empid	country	emphiredate
121034	US	01JAN2011
121088	US	01JAN2011
121146	US	01APR2010
120188	AU	01DEC2009
121124	US	01DEC2009
121032	AU	01MAR2010
120754	AU	01MAY2010
120193	AU	01SEP2009
120194	AU	01FEB2009
120277	US	01MAY2008

s105e01 sas

Figure 20: Evaluate the FROM Clause

2. Filter Rows Based on the WHERE Clause

The second phase filters rows based on the condition in the WHERE clause returning only rows for which the condition evaluates to true.

In this query, the WHERE filtering phase filters only rows for employees hired on or after January 1, 2009. 9 rows are returned from this phase and are provided as input to the next one.

Employee ID	country	Employee Hire Date
121034	US	01JAN2011
121088	US	01JAN2011
121146	US	01APR2010
120188	AU	01DEC2009
121124	US	01DEC2009
121032	AU	01MAR2010
120754	AU	01MAY2010
120193	AU	01SEP2009
120194	AU	01FEB2009

```
proc sql;
  SELECT empid, country,emphiredate
    FROM sgf2020.logicalq
      WHERE emphiredate >= '1jan2009'd;
quit;
```

s105e01

Figure 21: Filter Rows Based on the WHERE Clause

Typical Mistakes

A typical mistake made by not understanding the logical query processing is attempting to refer in the WHERE clause to a column alias defined in the SELECT clause. This isn't allowed because the WHERE clause is evaluated before the SELECT clause.

As an example, consider the following query.

```
proc sql;
  select country, YEAR(emphiredate) AS yearhired
    FROM SGF2020.logicalq
      WHERE yearhired >= 2009;
quit;
```

This query fails with the following error.

```
ERROR: The following columns were not found in the
contributing tables: yearhired.
```

s105e01

Figure 22: Typical Mistakes

If you understand that the WHERE clause is evaluated before the SELECT clause, you realize that this attempt is wrong because at this phase, the attribute yearhired **doesn't yet exist.** You can indicate the expression YEAR(employee_hire_date) >= 2009 in the WHERE clause.

3. Group Rows Based on the GROUP BY Clause

This phase defines a group for each distinct combination of values in the grouped elements from the input table.

It then associates each input row to its respective group. The query groups the rows by country and YEAR(employee_hire_date).

Within the 9 rows in the input table, this step identifies 5 groups. Here are the groups and the detail rows that are associated with them (redundant information removed for purposes of illustration).

Country	yearhired	numemp
AU	2009	3
AU	2010	2
US	2009	1
US	2010	1
US	2011	2

```
proc sql;
  SELECT country,YEAR(EMPHIREDATE) as yearhired, count(*) as
  numemp
    FROM sgf2020.logicalq
      WHERE emphiredate >= '1jan2009'd
        GROUP BY country, yearhired;
Quit;
```

s105e01

Figure 23: Group Rows Based on the GROUP BY Clause

Understanding the GROUP BY clause

group US 2009 has 1 detail row with employee 121124;
group US 2010 also has 1 detail row with employee 121146
group US 2011 has 2 detail rows with employees 121034 & 121088
The final result of this query has one row representing each group (unless filtered out).

Country	yearhired	numemp
AU	2009	3
AU	2010	2
US	2009	1
US	2010	1
US	2011	2

Country	year	empid	Country	emphiredate
AU	2009	120188	AU	01DEC2009
		120193	AU	01SEP2009
		120194	AU	01FEB2009

Country	year	empid	Country	emphiredate
AU	2010	121032	AU	01MAR2010
		120754	AU	01MAY2010

Country	year	empid	Country	emphiredate
US	2009	121124	US	01DEC2009

Country	year	empid	Country	emphiredate
US	2010	121146	US	01APR2010

Country	year	empid	Country	emphiredate
US	2011	121034	US	01JAN2011
		121088	US	01JAN2011

14

sas

Figure 24: Understanding the GROUP BY Clause

4. Filter Rows Based on the HAVING Clause

The Having clause filters data based on a condition, but is evaluated after data has been grouped. It is evaluated per group and filters groups as a whole.

The HAVING clause uses the condition COUNT(*) > 1, to filter only country and hire year groups with more than one employee.

```
proc sql;
  SELECT country,YEAR(EMPHIREDATE) as yearhired,
count(*) as numemp
    FROM sgf2020.logicalq
      WHERE emphiredate >= '1jan2009'd
        GROUP BY country, yearhired
          HAVING count(*) > 1;
Quit;
```

Country	yearhired	numemp
AU	2009	3
AU	2010	2
US	2009	1
US	2010	1
US	2011	2

only the groups AU(2009), AU(2010) and US(2011) qualify.

country	yearhired	numemp
AU	2009	3
AU	2010	2
US	2011	2

s105e01

sas

Figure 25: Filter Rows Based on the HAVI NG Clause

5. Process the SELECT Clause

The 5th phase is responsible for processing the SELECT clause.

Its interesting that this is the point in logical query processing where it gets evaluated—almost last. Also interesting considering the fact that the SELECT clause appears first in the query.

```
proc sql;
 SELECT country, YEAR(emphiredate) AS
 yearhired,COUNT(*) AS numemp
   FROM SGF2020.logicalq
     WHERE emphiredate >= "1jan2009"d
       GROUP BY country, yearhired
         HAVING COUNT(*) > 1
           ORDER BY country, yearhired DESC;
QUIT;
```

s105e01

sas

Figure 26: Process the SELECT Clause

5. Desired Output

country	yearhired	numemp
AU	2010	2
AU	2009	3
US	2011	2

Figure 27: The Desired Output

CONCLUSION

This paper attempted to showcase the best strengths of PROC SQL and lay out these strengths step-by-step. The author has used her teaching and consulting experiences to highlight those tips that are very unique to PROC SQL.

ACKNOWLEDGEMENTS

The author is grateful to the many SAS users that have entered her life. Charu is grateful to the SAS Global Forum User Committee for the opportunity to present this paper. She would also like to express her gratitude to her manager, Stephen Keelan, without whose support and permission, this paper would not be possible.

CONTACT INFORMATION

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

Charu Shankar
SAS Institute Canada, Inc.
Charu.shankar@sas.com
https://blogs.sas.com/content/author/charushankar/

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.

REFERENCES

SAS 9.4 SQL Procedure User's Guide
https://go.documentation.sas.com/?docsetId=sqlproc&docsetTarget=titlepage.htm&docsetVersion=9.4&locale=en

Logical Query Processing Order
"A database professional's best friend." Shankar, Charu
https://blogs.sas.com/content/sastraining/2013/02/04/a-database-professionals-best-friend-2/

PROC SQL Syntax Order
"Go home on time with these 5 PROC SQL tips**."** Shankar, Charu
https://blogs.sas.com/content/sastraining/2012/04/24/go-home-on-time-with-these-5-proc-sql-tips/

PROC SQL DICTIONARY Tables
"**Know** Thy Data: Techniques for Data Exploration.**" Shankar, Charu**
https://www.pharmasug.org/proceedings/2018/BB/PharmaSUG-2018-BB11.pdf

"**Working with Subquery in the SQL** Procedure." **Zhang, Lei**, and Yi, Danbo
https://www.lexjansen.com/nesug/nesug98/dbas/p005.pdf

Boolean in SQL
"#1 SAS programming tip for 2012." **Shankar, Charu**
https://blogs.sas.com/content/sastraining/2012/05/10/1-sas-programming-tip-for-2012/