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 FEW WORKERS GO HOME ON TIME
```

```
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 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>>;
QUIT;
```

**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.
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

```sql
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.
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

![Figure 5: Employee Counts by Department](image)

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

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

Business Scenario 3

Create a report that lists the following for each department:
• total number of managers
• total number of non-manager employees
• manager-to-employee (M/E) ratio

Below is a rough sketch of the desired report.

<table>
<thead>
<tr>
<th>Department</th>
<th>Managers</th>
<th>Employees</th>
<th>M/E Ratio</th>
</tr>
</thead>
<tbody>
<tr>
<td>Accounts</td>
<td>1</td>
<td>5</td>
<td>20%</td>
</tr>
<tr>
<td>Administration</td>
<td>2</td>
<td>20</td>
<td>10%</td>
</tr>
</tbody>
</table>

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.

FIND Function

The FIND function returns the starting position of the first occurrence of a substring within a string (character value).

Find the starting position of the substring "Manager" in the character variable Job_Title.

\[ \text{FIND}(\text{Job_Title}, \text{"manager"}, \text{"i"}) \]

The value returned by the FIND function is 16.
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

PROC SQL Output
Manager-to-Employee Ratios

<table>
<thead>
<tr>
<th>Department</th>
<th>Managers</th>
<th>Employees</th>
<th>Ratio</th>
</tr>
</thead>
<tbody>
<tr>
<td>Accounts</td>
<td>3</td>
<td>14</td>
<td>21.4%</td>
</tr>
<tr>
<td>Accounts Management</td>
<td>1</td>
<td>8</td>
<td>12.5%</td>
</tr>
<tr>
<td>Administration</td>
<td>6</td>
<td>59</td>
<td>17.3%</td>
</tr>
<tr>
<td>Connection</td>
<td>1</td>
<td>10</td>
<td>10.0%</td>
</tr>
<tr>
<td>Engineering</td>
<td>1</td>
<td>8</td>
<td>12.5%</td>
</tr>
<tr>
<td>Executives</td>
<td>0</td>
<td>4</td>
<td>0.0%</td>
</tr>
<tr>
<td>Group Financials</td>
<td>0</td>
<td>3</td>
<td>0.0%</td>
</tr>
<tr>
<td>Group HR Management</td>
<td>3</td>
<td>10</td>
<td>20.0%</td>
</tr>
<tr>
<td>IS</td>
<td>2</td>
<td>23</td>
<td>8.7%</td>
</tr>
<tr>
<td>Logistics Management</td>
<td>8</td>
<td>9</td>
<td>76.0%</td>
</tr>
<tr>
<td>Marketing</td>
<td>6</td>
<td>14</td>
<td>42.9%</td>
</tr>
<tr>
<td>Purchasing</td>
<td>3</td>
<td>10</td>
<td>30.0%</td>
</tr>
<tr>
<td>Sales</td>
<td>0</td>
<td>201</td>
<td>0.0%</td>
</tr>
<tr>
<td>Sales Management</td>
<td>5</td>
<td>60</td>
<td>83.3%</td>
</tr>
<tr>
<td>Secretary of the Board</td>
<td>0</td>
<td>2</td>
<td>0.0%</td>
</tr>
<tr>
<td>Stock &amp; Shipping</td>
<td>5</td>
<td>21</td>
<td>20.0%</td>
</tr>
<tr>
<td>Strategy</td>
<td>0</td>
<td>2</td>
<td>0.0%</td>
</tr>
</tbody>
</table>

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.

```sql
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.

```sql
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
```

![Figure 10: PROC SQL Output Tables Containing an ID Column](image)

All ID column names are stored in uniform uppercase, so the UPCASE function is not needed the next time that a query such as this is executed.

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.

```sql
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
```
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.

PROC SQL supports two types of joins: inner joins return only matching rows.

Outer joins return all matching rows, plus nonmatching rows from one or both tables.

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

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**

```sql
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.
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.

---

**Figure 16: Return the Employee’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.

---

**Figure 17: Determine the ID of the Employee’s Manager**
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.

```sql
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

---

**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.

```sql
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

---

**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.

```sql
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;
```

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.

```
proc sql;
  SELECT empid, country, emphiredate
  FROM sqf2020.logicalq
  WHERE emphiredate >= '1jan2009'd;
quit;
```

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 sqf2020.logicalq
  WHERE yearhired >= 2009;
quit;
```

This query fails with the following error.

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

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.

Here are the groups and the detail rows that are associated with them ( redundant information removed for purposes of illustration).

```
proc sql;
  SELECT country, YEAR(emphiredate) as yearhired, count(*) as numemp
  FROM sqf2020.logicalq
  WHERE emphiredate >= '1jan2009'd
  GROUP BY country, yearhired;
quit;
```

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).

<table>
<thead>
<tr>
<th>Country</th>
<th>Year hired</th>
<th>Num emp</th>
</tr>
</thead>
<tbody>
<tr>
<td>US</td>
<td>2009</td>
<td>1</td>
</tr>
<tr>
<td>US</td>
<td>2010</td>
<td>1</td>
</tr>
</tbody>
</table>

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 it 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.

```sql
proc sql;
SELECT country, YEAR(EMPHIREDATE) AS yearhired,
count(*) AS numemp
FROM sgp2020.logicalq
WHERE emphiredate >= '1jan2009'd
GROUP BY country, yearhired
HAVING count(*) > 1;
QUIT;
```

Figure 25: Filter Rows Based on the HAVING 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.

```sql
proc sql;
SELECT country, YEAR(emphiredate) AS
yearhired,COUNT(*) AS numemp
FROM sgp2020.logicalq
WHERE emphiredate >= "1jan2009"d
GROUP BY country, yearhired
HAVING COUNT(*) > 1
ORDER BY country, yearhired DESC;
QUIT;
```

Figure 26: Process the SELECT Clause
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

“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/