Paper 111-2022

Your Query's No Good Here: PROC SQL Code That Doesn't Work Outside of SAS

Jedediah Teres, MDRC

ABSTRACT

PROC SQL is based on the 1992 ANSI standard. As a result, SQL implementations based on more recent standards have functions not available in PROC SQL, like the ordered analytic family of functions (also known as "window" functions). Conversely, there are enhancements and extensions to SQL available in PROC SQL that are unique to SAS, such as the OUTER UNION CORRESPONDING operator. This paper describes scenarios in which valid PROC SQL will not work in other SQL implementations such as SQL Server and proposes alternative solutions.

INTRODUCTION

The SQL procedure in SAS is a useful complement to both the DATA step and other procedures such as PROC MEANS or PROC FREQ. Some programming tasks can be done more easily with PROC SQL than with Base SAS, and other tasks would be nearly impossible without PROC SQL (Hu 2004). Though some consider PROC SQL an advanced topic in SAS programming, almost everything is based on the SELECT statement, and from that starting point, many complex programming tasks can be performed with relatively few lines of code (Ronk, First, and Beam 2002).

PROC SQL is based on the 1992 SQL standard (Andrews and Miyani 2011), while most other implementations of SQL such as Oracle, SQL Server, Teradata, PostgreSQL, MySQL, and SQLite are based on the 1999 or newer standards.

For some SAS users, PROC SQL is the only experience they have working with SQL. While PROC SQL can provide a strong foundation in the use of SQL more generally, there are some features that SAS has added to PROC SQL that will not be found in other implementations. The PROC SQL explicit pass-through facility allows SQL code written for a specific relational database management system (RDBMS) to be run in the database, which can be more efficient if it means less data being transferred from the database to SAS (Capobianco 2011).

This paper is aimed at SAS users with experience using PROC SQL who are interested in using other implementations of SQL in the pass-through facility. Several examples of useful queries that will not run outside of SAS are presented, along with alternative code to make the queries compatible with most other implementations of SQL. The examples contained herein were written in SAS 9.4. The database used is a SQL Server database running version 14.

THE DATA USED IN THIS PAPER

The data featured in the examples presented in this paper is structured like quarterly earnings data reported from a state unemployment agency. Data of this type are often used at MDRC for evaluations of jobs programs. There are four columns—participant SSN (SSN), year, quarter, employer ID (EmpID), and wages. These data were randomly generated for training purposes and a sample is shown in **Error! Reference source not found.**.

SSN	year	quarter	EmpID	wages
193605824	2017	2	241936058	$9,022.00
193605824	2017	3	824193605	$1,474.00
193605824	2017	4	824193605	$11,232.00
411419376	2017	1	376411419	$523.00
411419376	2017	3	376411419	$2,384.00
411419376	2018	2	376411419	$7,794.00
509525480	2017	1	480509525	$2,810.00
509525480	2017	3	480509525	$6,043.00

SSN	year	quarter	EmpID	wages
509525480	2017	4	480509525	$12,344.00
509525480	2018	1	480509525	$13,140.00
509525480	2018	2	480509525	$2,450.00

Table 1: A Sample of the Data Used

SAS-SPECIFIC SQL

To highlight differences between PROC SQL and T-SQL (the SQL implementation of SQL Server), T-SQL will be used in explicit pass-through alongside PROC SQL code pointing to the same source table. Explicit pass-through is when the programmer writes SQL code to be passed directly to the RDBMS to be run where the data is stored and uses the specific SQL language of the RDBMS used (Hampton 2011). Processing the data in-database can be advantageous if it avoids transferring large amounts of data, or when the RDBMS has SQL functions not available in SAS (Capobianco 2011).

OUTER UNION CORRESPONDING

One of the SAS enhancements to the 1992 SQL standard is the OUTER UNION CORRSPONDING set operator. The OUTER UNION set operator stacks two query results without aligning common columns; the CORRESPONDING option aligns common columns (Schreier 2006).

In practice, OUTER UNION CORRESPONDING is used to stack two query results with at least one non-overlapping column, as shown below:

```
proc sql ;
     select    quarter,
               year,
               sum(wages) as sum_of_wages format = dollar20.
     from      train_db.UI_v
     group by  quarter,
               year
     outer union corresponding
     select    year,
               sum(wages) as sum_of_wages format = dollar20.
     from      train_db.UI_v
     group by  year
     order by  year,
               missing(quarter),
               quarter ;
quit ;
```

The result of this query is shown in Table 2.

quarter	year	sum_of_wages
1	2017	$4,269,923
2	2017	$4,097,679
3	2017	$3,928,737
4	2017	$3,849,919
.	2017	$16,146,258
1	2018	$3,967,115
2	2018	$4,118,987
3	2018	$4,095,330
4	2018	$4,061,610
.	2018	$16,243,042
1	2019	$3,824,604
2	2019	$3,992,729
3	2019	$3,919,977
4	2019	$3,867,588

quarter	year	sum_of_wages
.	2019	$15,604,898
1	2020	$4,079,806
2	2020	$4,235,228
3	2020	$4,061,562
4	2020	$4,131,821
.	2020	$16,508,417
1	2021	$3,884,163
.	2021	$3,884,163

Table 2: OUTER UNION CORRESPONDING Result

Unfortunately, OUTER UNION is a SAS enhancement to standard SQL. This is the error that results from trying use OUTER UNION CORRESPONDING in the SQL Server environment:

```
ERROR: CLI describe error: [Microsoft][ODBC Driver 17 for SQL Server][SQL
Server]Incorrect syntax near the keyword 'outer'. :
       [Microsoft][ODBC Driver 17 for SQL Server][SQL Server]Statement(s) could not be
prepared.
```

To create result sets like those produced using OUTER UNION CORRESPONDING, users will have to add columns with NULL values as placeholders. Note the use of CONNECT and DISCONNECT statements, as well as SELECT * FROM CONNECTION TO. This is explicit pass-through code:

```
proc sql ;
     ** Connect to SQL Server using CONNECT TO statement ;
     connect to odbc as training (dsn = 'training') ;

     select    *
     from      connection to training (
               select    quarter,
                         year,
                         avg(wages) as avg_wages
               from      UI_v
               group by  quarter,
                         year
               union
               select    null as quarter,
                         year,
                         avg(wages) as avg_wages
               from      UI_v
               group by  year
               )
     order by  year,
               missing(quarter),
               quarter ;
     ** Disconnect from the SQL Server ;
     disconnect from training ;
quit ;
```

The OUTER UNION CORRESPONDING set operator is an SAS-specific enhancement to the SQL implementation based on the 1992 standard. The 1999 standard introduced the GROUPING SETS subclause of the GROUP BY clause:

```
proc sql ;
     ** Connect to SQL Server using CONNECT TO statement ;
     connect to odbc as training (dsn = 'training') ;

     select    *
     from      connection to training (
                /* native T-SQL (SQL Server) code */
                select     quarter,
                           year,
                           avg(wages) as avg_wages
                from       UI_v
                group by   grouping sets (
                                          (quarter, year),
                                          year)
                order by   year,
                           isnull(quarter, 5)
                );

          ** Disconnect from the SQL Server ;
          disconnect from training ;
quit ;
```

While this code cannot replace all possible use cases for OUTER UNION CORRESPONDING, the GROUPING SETS subclause is very powerful and useful to know for coders working with other implementations of SQL, whether directly or via explicit pass-through.

One thing to note—the MISSING function is native SAS code. To sort the results using SQL Server, the T-SQL function ISNULL can be used. The ISNULL function takes two arguments—the first is a column or expression, and the second is the value to return when the expression is null. In this case, “ISNULL(quarter, 5)” returns a value of 5 when quarter is null, and the original value of “quarter” otherwise. This substitution makes the code less reliant on native SAS code and does the sorting in-database.

IN-LINE VIEWS

In-line VIEWs (sometimes erroneously referred to as subqueries) are SELECT statements contained in the FROM clause. This example, which summarizes the number of IDs by the number of observations, contains an in-line VIEW:

```
proc sql ;
     select     NObs,
                count(*) as NIDs
     from (     select     SSN,
                           count(*) as NObs
                from       train_db.UI_v
                group by   SSN)
     group by   NObs
     order by   NObs ;
quit ;
```

The result of this query is shown in Table 3.

NObs	NIDs
3	1
4	9
5	26
6	55
7	82
8	98
9	165
10	147
11	139
12	85
13	63

NObs	NIDs
14	26
15	10
16	4
17	2

Table 3: In-Line VIEW Result

These are allowed in other implementations of SQL, but they must be given an alias, even is the alias is not used or referenced. The resulting error is not helpful for troubleshooting because it highlights the word “group” from the GROUP BY clause:

```
ERROR: CLI describe error: [Microsoft][ODBC Driver 17 for SQL Server][SQL
Server]Incorrect syntax near the keyword 'group'. :
       [Microsoft][ODBC Driver 17 for SQL Server][SQL Server]Statement(s)
could not be prepared.
```

A simple fix is to add an alias, as shown here:

```
proc sql ;
     ** Connect to SQL Server using CONNECT TO statement ;
     connect to odbc as training (dsn = 'training') ;

     select    *
     from      connection to training (
               /* native T-SQL (SQL Server) code */
               select    NObs,
                         count(*) as NIDs
               from (    select    SSN,
                                   count(*) as NObs
                         from      UI_v
                         group by  SSN) as a
               group by  NObs
               )
     order by  NObs ;

     ** Disconnect from the SQL Server ;
     disconnect from training ;
quit ;
```

Another feature of the 1999 SQL standard is the Common Table Expression (CTE). For SAS users, it might be helpful to think of a CTE like a SAS macro or macro variable.

Common Table Expressions are declared using the WITH[1] keyword as shown below. The definition follows in parentheses. The use of the underscore in the naming is not required, but it might be helpful for recognizing a reference to a CTE in subsequent code:

```
proc sql ;
     ** Connect to SQL Server using CONNECT TO statement ;
     connect to odbc as training (dsn = 'training') ;

     select    *
     from      connection to training (
               /* native T-SQL (SQL Server) code */
               /* use WITH to create CTE-- note underscore in name */
               with      _ObsCount as (
                         select    SSN,
                                   count(*) as NObs
```

[1] The syntax coloring of the WITH keyword was added by the author and does not appear in SAS editors.

```
                    from      UI_v
                    group by  SSN
                    )

          select    NObs,
                    count(*) as NIDs
          from      _ObsCount
          group by  NObs
          )
     order by  NObs ;

     ** Disconnect from the SQL Server ;
     disconnect from training ;
quit ;
```

This technique can replace nested SELECT statements and make code easier to read.

REMERGING SUMMARY STATISTICS

The next several examples are all instances in which the following note appears in the log:

```
NOTE: The query requires remerging summary statistics back with the
original data.
```

As it is neither a WARNING nor ERROR, it might escape the programmer's notice. But it is a clear indication that the PROC SQL code will not work in the native SQL implementation of the target RDBMS.

Adding One Value to All Rows

Warren (2007) describes a technique to add one value to all rows of a dataset using PROC SQL. A variation on that approach is shown here:

```
proc sql ;
     select    *,
               avg(wages) as avg_wages format = dollar20.2
     from      train_db.UI_v) ;
quit ;
```

A sample of the result is shown below in Table 4. Note the value of avg_wages is constant across rows.

SSN	year	quarter	EmpID	wages	avg_wages
411419376	2017	1	376411419	$523.00	$8,175.10
509525480	2017	1	480509525	$2,810.00	$8,175.10
661634944	2017	3	944661634	$10,884.00	$8,175.10
580629904	2017	4	904580629	$11,253.00	$8,175.10
514925816	2018	1	816514925	$12,823.00	$8,175.10
849746648	2018	1	648849746	$913.00	$8,175.10
193605824	2021	1	824193605	$13,762.00	$8,175.10
487924136	2021	1	136487924	$6,411.00	$8,175.10
235008400	2019	3	002350084	$15,600.00	$8,175.10
668835392	2018	3	392668835	$6,772.00	$8,175.10

Table 4: One Value Added to All Rows

Running this code in explicit pass-through results in the following error:

```
ERROR: CLI describe error: [Microsoft][ODBC Driver 17 for SQL Server][SQL
Server]Column 'UI_v.SSN' is invalid in the select list
       because it is not contained in either an aggregate function or the GROUP BY
clause. : [Microsoft][ODBC Driver 17 for SQL
       Server][SQL Server]Statement(s) could not be prepared.
```

There are two things to note. The first is that columns that are not contained in either an aggregate function or GROUP BY statement are not allowed. The second is that the specific error named a column that we did not specify.

The Feedback Option

The FEEDBACK option will write to the LOG the actual syntax that is submitted to be run. In this example, the NOEXEC option is used as well to prevent the code from executing:

```
proc sql feedback noexec ;
     select    *,
               avg(wages) as avg_wages format = dollar20.2
     from      train_db.UI_v ;
quit ;
```

The log show the following:

```
NOTE: Statement transforms to:

        select UI_v.SSN, UI_v.year, UI_v.quarter, UI_v.EmpID, UI_v.wages,
AVG(UI_v.wages) as avg_wages format=DOLLAR20.2
          from TRAIN_DB.UI_v ;
```

Referring to the prior ERROR message, the column UI_v.SSN was mentioned. It is the first column listed in the submitted code.

There are two possible solutions. One is to simply use a CROSS JOIN to add the value to all rows:

```
proc sql ;
     ** Connect to SQL Server using CONNECT TO statement ;
     connect to odbc as training (dsn = 'training') ;

     select    *
     from      connection to training (
               /* native T-SQL (SQL Server) code */
               select    a.*,
                         b.avg_wages
               from      UI_v as a
                         cross join
                    (    select    avg(wages) as avg_wages
                         from      UI_v) as b
               ) ;

     ** Disconnect from the SQL Server ;
     disconnect from training ;
quit ;
```

An alternative approach makes use of a window function.

The ordered analytic and window functions are part of the 1999 SQL standard. Andrews and Miyani (2011) do an excellent job of describing the syntax for these functions and comparing the results to SAS code. A deep dive into these functions is beyond the scope of this paper, but their paper is very highly recommended.

In this example, the OVER clause is used with the AVG function to take the average value of all rows:

```
proc sql ;
     ** Connect to SQL Server using CONNECT TO statement ;
     connect to odbc as training (dsn = 'training') ;

     select    *
     from      connection to training (
                /* native T-SQL (SQL Server) code */
                select     *,
                           avg(wages) over() as avg_wages
                from       UI_v
                ) ;

     ** Disconnect from the SQL Server ;
     disconnect from training ;
quit ;
```

This does not require any joins or CTEs, and opens the door to further exploration of these window functions.

Selecting Rows Based on Comparisons to Aggregate Values

PROC SQL can be used to select rows based on their relationship to an aggregate value. For example, this query returns all rows with wages equal to the maximum value for wages:

```
proc sql ;
     select     *
     from       train_db.UI_v
     having     wages = max(wages) ;
quit ;
```

There are three quarters in which someone earned the maximum wages, as shown in Table 5.

SSN	year	quarter	EmpID	wages
158503640	2017	1	401585036	$26,000.00
721938696	2020	4	967219386	$26,000.00
360116184	2019	2	843601161	$26,000.00

Table 5: Maximum Wages from HAVING

When this code is run via explicit pass-through, we are told that this is not an acceptable use of the HAVING clause:

```
ERROR: CLI describe error: [Microsoft][ODBC Driver 17 for SQL Server][SQL
Server]Column 'UI_v.wages' is invalid in the HAVING
       clause because it is not contained in either an aggregate function or the GROUP
BY clause. : [Microsoft][ODBC Driver 17 for
       SQL Server][SQL Server]Statement(s) could not be prepared.
```

Two previous techniques can be combined to great effect in this case. By using a window function inside a CTE, we can change the HAVING clause to a WHERE clause and get the same result:

```
proc sql ;
     ** Connect to SQL Server using CONNECT TO statement ;
     connect to odbc as training (dsn = 'training') ;

     select     *
     from       connection to training (
                /* native T-SQL (SQL Server) code */
                with       _MW as (
                           select     *,
                                      max(wages) over() as max_wages
                           from       UI_v)

                select     *
                from       _MW
                where      wages = max_wages
                ) ;
     ** Disconnect from the SQL Server ;
     disconnect from training ;
quit ;
```

Identifying Duplicates and Other N-Tuples

Variations on this query are very useful for identifying rows with duplicate values. This is more flexible than using PROC SORT-based options like NODUPKEY or NODUPREC because we can GROUP BY different subsets of columns and thus are not limited to exact duplicates. In this case, we are looking for the SSNs with fewer than 5 observations:

```
proc sql ;
     select    *
     from      train_db.UI_v
     group by  SSN
     having    count(*) lt 5
     order by  SSN,
               year,
               quarter ;
quit ;
```

The result of this query shown in Table 6, which is a subset of all SSNs with 4 or fewer records; the remaining rows were omitted to save space.

SSN	year	quarter	EmpID	wages
122501400	2018	4	400122501	$11,484.00
122501400	2019	2	400122501	$7,741.00
122501400	2019	4	400122501	$8,724.00
122501400	2020	2	400122501	$12,683.00
218807392	2017	3	392218807	$11,653.00
218807392	2018	3	392218807	$12,550.00
218807392	2019	3	392218807	$1,091.00
218807392	2020	1	392218807	$7,880.00
460022400	2017	1	400460022	$10,453.00
460022400	2017	2	400460022	$1,793.00
460022400	2019	3	400460022	$1,742.00
819144744	2021	1	744819144	$2,673.00
915450736	2017	2	736915450	$13,724.00
915450736	2018	2	736915450	$11,923.00
915450736	2020	3	736915450	$10,374.00
915450736	2020	4	369154507	$14,352.00

Table 6: SSNs with Fewer Than 5 Observations (sample)

However, this code does not run in the native SQL Server environment:

```
ERROR: CLI describe error: [Microsoft][ODBC Driver 17 for SQL Server][SQL
Server]Column 'UI_v.year' is invalid in the select list
       because it is not contained in either an aggregate function or the GROUP BY
clause. : [Microsoft][ODBC Driver 17 for SQL
       Server][SQL Server]Statement(s) could not be prepared.
```

To get the desired result, we can expand on the techniques we have been building. In this case, we define a CTE with a window function and change the HAVING clause to a WHERE clause. The difference is that we are using a PARTITION in the OVER clause. The PARTITION is on SSN, which means that a row count will be created for each distinct value of SSN.

```
proc sql ;
     ** Connect to SQL Server using CONNECT TO statement ;
```

```
          connect to odbc as training (dsn = 'training') ;

          select    *
          from      connection to training (
                    /* native T-SQL (SQL Server) code */
                    with _innerQ as (
                         select    *,
                                   count(*) over(partition by SSN) as SSN_Count
                         from      UI_v)
                    select    *
                    from      _innerQ
                    where     SSN_Count < 5
                    ) ;

          ** Disconnect from the SQL Server ;
          disconnect from training ;
     quit ;
```

CONCLUSION

The SQL procedure offers flexible and complementary functionality to the DATA step and other procedures such as PROC MEANS and PROC FREQ. PROC SQL is based on the 1992 standard, though it has non-standard enhancements, and many SAS functions can be used in PROC SQL queries. Some of these SAS-specific enhancements mean that queries that run in PROC SQL will not work outside of SAS. Explicit pass-through can be very helpful; writing native SQL to leverage the full power of an available RDBMS can be more efficient. This paper presented several alternatives and introduced some SQL functionality that SAS users might have missed. While the code provided is T-SQL code for use with SQL Server, these queries can be run in most modern SQL implementations with only minor adjustments.

REFERENCES

Andrews, R and Miyani, J. 2011. "Teradata® for the SAS® Programmer: Ordered Analytical Functions, Hash Objects, ANSI SQL: 2003." *Proceedings of the SAS Global Forum*, 019-2011. Las Vegas, NV.

Capobianco, F. 2011. "Explicit SQL Pass-Through: Is It Still Useful?" *Proceedings of the SAS Global Forum*, 019-2011. Las Vegas, NV.

Hampton, J. 2011. "SQL Pass-Through and the ODBC Interface." *Proceedings of the Northeast SAS Users Group Conference*, PS04. Portland, ME.

Hu, W. 2004. "Top Ten Reasons to Use PROC SQL." *Proceedings of the SAS Users Group International Conference*, 042-29. Montreal, QC.

Lafler, K.P. 2017. "Advanced Programming Techniques with PROC SQL." *Proceedings of the SAS Global Forum*, 0930-2017. Orlando, FL.

Ronk, K.M., First, S., and Beam, D. 2002. "An Introduction to PROC SQL®." *Proceedings of the SAS Users Group International Conference*, 191-27. Orlando, FL.

Schreier, Howard. 2006. "SQL Set Operators: So Handy Venn You Need Them." *Proceedings of the SAS Users Group International Conference*, 242-31. San Francisco, CA.

Warren, A. 2007. "Adding One Value to All Observations." *Proceedings of the Northeast SAS Users Group Conference*, CC45. Baltimore, MD.

CONTACT INFORMATION

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

Jedediah Teres
jed.teres@gmail.com