## Paper 7540-2016

# PROC SQL for SQL Die-hards

Barbara Ross, Subprime & Underbanked Analytics Association LLC; Mary Jessica Bennett, Snap Finance

#### **ABSTRACT**

Inspired by Christianna William's paper on transitioning to PROC SQL from the DATA step, this paper aims to help SQL programmers transition to SAS by using PROC SQL. SAS® adapted the Structured Query Language (SQL) by means of PROC SQL back with Version 6. PROC SQL syntax closely resembles SQL, however, there some SQL features that are not available in SAS. Throughout this paper, we will outline common SQL tasks and how they might differ in PROC SQL; as well as introduce useful SAS features that are not available in SQL. Topics covered are appropriate for novice SAS users.

### INTRODUCTION

Transitioning from a SQL shop to a SAS shop, or vice versa, can be a challenging experience. Most SQL die-hards believe the PROC SQL procedure will be a life-saver. That is, until they realize that minor syntax differences between the two can turn into a major nuance. Contrary to what a SQL die-hard visualized, most of the time they will not be able to copy and paste SQL code into SAS by means of PROC SQL. In this paper, we outline a few of the issues that we, as programmers, encountered when deemed with the task of transforming SQL logic into SAS code. This list is not all encompassing, but includes some of the key issues we have been faced with while making the transition.

### **EXAMPLE 1. NOT EQUALS SHORTCUT**

One small change between SAS and SQL is the shortcut notation for not equals. In SQL, you can type "!=" and "<>". This notation is not recognized in SAS, you must use "^=". In both systems you can type out "NOT EQUAL" as well.

```
SQL Code:
SELECT *
FROM inputds
WHERE date_created != '2014-01-05';

SAS Code:
SELECT *
FROM inputds
WHERE date created ^= '05jan2014'd;
```

### **EXAMPLE 2. CONCATENATING STRINGS**

With MSSQL, a plus sign, "+", can be used to concatenate strings. Similarly in SAS, one can use double pipes, "||". Outside of the notations, there several functions to concatenate strings as well. The SAS CAT and SQL CONCAT functions are very similar and both work like the "+" and "||" notations. SAS also has variations on the CAT function that automatically trim leading and trailing blanks, removing the need to use RIGHT/LEFT/TRIM functions. Popular SAS concatenation functions include: CATX (removes trailing and leading blanks and inserts delimiters), CATT (removes trailing blanks), and CATS (removes trailing and leading blanks).

```
FROM Customers;

SAS Code - "||" & CATX:

SELECT FirstName || ' ' || LastName As FullName
, CATX(' ', FirstName, LastName) As FullName2

FROM Customers;
```

## **EXAMPLE 3. SELECTING N OBSERVATIONS**

When exploring a new database, it is often helpful to be able to select a limited number of records from a dataset. With SAS, you will use the OUTOBS= statement to limit the number of records shown. The equivalent in MSSQL is the TOP statement, and MySQL uses the LIMIT statement.

## **EXAMPLE 4. VARIABLE NAME LENGTH**

It's worth noting that SAS is much more conservative than SQL when it comes to naming variables. If you are accessing a SQL database through a SAS libname, SAS will crop/rename your variable names to conform to their naming conventions.

Allowed variable name lengths:

```
SAS - 32 characters
Oracle - 30 characters
MSSQL - 128 characters (116 characters in temporary tables)
MySQL - 64 characters
```

Also SAS does not support variable names that start with a number. In SQL, you can't declare a name beginning with numbers, but you can rename them to. When accessing that table in SAS, SAS will prefix the name with an underscore (i.e. '2paydefault' renamed to '2paydefault').

SQL allows for variable names with spaces as well. SAS Enterprise Guide (EG) also allows spaces, so these variables will not be renamed when accessing your SQL tables through SAS EG. However, it is difficult to reference variables with spaces in their names. To have SAS rename variables names without spaces, you can use the system option VALIDVARNAME. Setting this to v7 will replace the spaces in the variable names with underscores ("\_") making them easier to work with in SAS.

```
SAS OPTIONS VALIDVARNAME=v7;
```

## **EXAMPLE 5. DATA TYPES**

SQL has over 30 different data types that can be used when defining variables. SAS only has 2 data types, character and numeric (dates are numeric). So during your conversion, you will need edit any code that CASTs or CONVERTs your data to types not available in SAS (e.g. varchar, integer, decimal, float).

SQL Code:

```
SELECT CAST(Ndefaulted as float)/Nloans AS Default_Rate FROM inputds;

SAS Code:
SELECT Ndefaulted/Nloans AS Default_Rate FORMAT=3.1
FROM inputds;
QUIT;
```

To control how data is displayed, SAS assigns formats to each variables. There are hundreds of ready-to-use formats, but SAS also allows programmers to create their own. Custom formats can also be used to recode values as an alternative to CASE WHEN statements. It's important to note that formats do not change the actual value of the data. If you want to permanently change a value, the format will need to be applied to a variable using the PUT function.

The example below shows how a format can be used to change a value in SAS, replacing the CASE WHEN statement.

To control storage allotted, in SAS each variable is associated with a length. Lengths are automatically assigned by SAS unless explicitly defined before the data is read. They are set to the length of the first record read in. For example, if the first record in your data set has a value of "No", SAS will set the length of that variable to 2. If the next record is "Yes", SAS will truncate that record to have a length of 2, therefore keeping "Ye". A nice thing about PROC SQL is that SAS will set the length of a new variable created by CASE WHEN to the longest length possible by the statement (e.g. CASE WHEN 1 THEN 'No' ELSE 'Yes' END will have a length of 3).

Working together, the data type, length, and format are what defines how a variable is treated, stored and displayed (respectively) in SAS. If you want to know more about SAS lengths and formats or SQL data types and implicit conversions, there are a few links referenced in the recommended reading section.

### **EXAMPLE 6. CONVERTING NUMERIC TO CHARACTER**

To convert a variable to a different data type, SQL has the CONVERT and CAST functions. SAS does not support CONVERT/CAST rather you must use the PUT and INPUT functions. The PUT function applies a format to a value, which allows it to convert a numeric value to a character field. INPUT reads in a value based on an informat, which allows you to read in a character field as numeric (Just remember IN for INput and INformat!). Note you must create a new variable if changing the data type. In PROC SQL, you can name the variable the same rendering it with an AS statement, but if converting in a SAS DATA step you must rename the original variable if you want the name to remain the same for the converted field.

#### SYNTAX:

```
PUT(source, format)
INPUT(source, informat)
CONVERT(data type, expression)
```

In the example below, SSN is numeric and amount is character. The z9 format adds leading zeros so that the outputted SSN value has a length of 9 digits (ex. 1234567 to 001234567). The dollar20 informat will read the numeric value of amount, ignoring the dollar sign and commas (ex. \$25.43 to 25.43).

```
SQL Code:
```

```
SELECT RIGHT('00000000' + CONVERT(VARCHAR(9), SSN), 9) as SSN
, CONVERT(float, REPLACE(REPLACE(amount,',', ''),'$','')) as
amount
FROM inputds;

SAS Code - No need to rename with PROC SQL:
PROC SQL;
CREATE TABLE new AS
SELECT PUT(SSN, z9.) as SSN
, INPUT(amount, dollar20.) as amount
FROM inputds;
QUIT;

SAS Code - Renaming with DATA Step:
DATA new;
SET inputds (RENAME=(ssn=ssnn amount=amountc);
SSN = PUT(SSNn, z9.);
Amount = INPUT(amountc, dollar20.);
```

## **EXAMPLE 7. JOINING ON DIFFERENT DATA TYPES**

DROP ssnn amountc;

Another fun tidbit that will have you banging your head on your desk is that SQL is very liberal with implicit data type conversions and automatically forces a merge when you compare different data types. SAS only has two data types (character and numeric) and they typically require you to explicitly convert the variables to match. To do this, you need to convert the character variables to numeric by using the INPUT function or the numeric variables to character with the PUT function.

In the example below, inputds1 cust id is character and inputds2 cust id is numeric.

WHERE inputds1.cust id = PUT(inputds2.cust id,\$5.);

#### SQL Code:

RUN;

```
SELECT *
FROM inputds1, inputds2
WHERE inputds1.cust_id = inputds2.cust_id;

SAS Code - Convert both to character:
SELECT *
FROM inputds1, inputds2
```

### SAS Code – Convert both to numeric:

```
SELECT *
FROM inputds1, inputds2
WHERE INPUT(inputds1.cust_id, best32.) = inputds2.cust_id;
```

## **EXAMPLE 8. SETTING TOGETHER SIMILAR DATASETS**

Another perk of converting your code to SAS is that the task of combining many datasets can be done quickly. In SQL, this can be tedious as it requires you to select the variables desired from each dataset and join them with a UNION statement. Within SAS, PROC SQL functions the same way as SQL. In this case it is actually easiest to use the SAS DATA step. With the DATA step, you can simply list the datasets to be merged with the SET statement. The SAS DATA step also has a shortcut for datasets with similar root names. Simply place a colon ":" after the root of the name and SAS will set together all the datasets that begin with that name.

```
SQL Code:
SELECT *
FROM inputds 201411
UNION
SELECT *
FROM inputds 201412
SAS Code – PROC SQL:
PROC SQL;
      CREATE TABLE alldata AS
      SELECT *
      FROM inputds 201411
UNION
      SELECT *
      FROM inputds 201412;
QUIT;
SAS Code – DATA Step:
DATA alldata;
      SET inputds 201411 inputds 201412;
RUN;
SAS Code - DATA Step Shortcut Method:
DATA alldata;
      SET inputds:;
RUN;
```

### **EXAMPLE 9. USING DATE and DATETIME VALUES**

Big difference between SQL and SAS is how you treat date and datetime variables. In SQL if you declare a date and do not add a time to the value, it will default to 0:0:0. This means you can compare date variables with datetime variables without having to convert between the two types. In SAS, variables must be both date or datetime to accurately compare them. This is because SAS stores dates/datetimes as the number of days/seconds respectively since Jan 1, 1960. When working with a datetime value in SAS, you must be sure to convert it to a date before comparing to another date variable or you will get an incorrect result.

In the example below, date\_created is a datetime variable.

```
SQL Code:
SELECT *
FROM inputds
WHERE date_created >= '2014-01-05'

SAS Code:
SELECT *
FROM inputds
```

```
WHERE DATEPART(date created) >= '05jan2014'd;
```

#### **EXAMPLE 10. INCREMENTING DATES**

To increment a date in SQL, the DATEADD and DATEDIFF functions are commonly used. The DATEADD function takes a date value and increments it by a user specified interval. The DATEDIFF function is used to calculate the number of intervals between two dates. These functions are not available in SAS. SAS uses the INTCK and INTNX functions instead.

The INTNX function is similar to DATEADD and is useful for determining the start and end points of a period of time. INTCK is like DATEDIFF in that it returns the number of intervals between two date or datetime values. Fun way to remember them is the C in INTCK is for Count (as in count the days), and the N in INTNX is for Next (as in next date)!

### Syntax:

## **EXAMPLE 11. SELECTING ROWS BASED ON A CREATED VARIABLE**

SAS gives programmers the ability reference a calculated field in a WHERE clause. This is unique to PROC SQL and is not available in SQL. To do this, you simply add 'CALCULATED' before referencing the variable. The long way in SQL requires you to type out the whole calculation to filter on it.

#### SQL Code:

## **EXAMPLE 12. NUMBER RECORDS WITHIN A GROUP**

MSSQL programmers have the RANK function available to them in order to number records within a group. This ranking is then often used to dedup datasets by specifying the first observation to keep. There is no clear substitute for the RANK function in MySQL or SAS. In MySQL, programmers can update temporary variables to number records within a group. Whereas SAS may not have a RANK equivalent, there are several native SAS procedures (outside of PROC SQL) to solve these issues. To dedup a dataset in SAS, you can use the SORT Procedure and to number records within a group, the SAS DATA step first, processing can be used.

```
MSSQL Code:
SELECT a.*
       ,RANK() over (partition by cust id order by date created) as n
FROM inputds a
ORDER BY cust id, date created
MySQL Code:
SET @curRow=0;
SELECT a.*
      , (CASE cust id WHEN @curld THEN @curRow := @curRow +1
       ELSE @curRow :=1 and @curId := cust id END) +1 as n
FROM inputds a
ORDER BY cust id, date created
SAS Code – Dedup dataset:
*Sort first so the record you want to keep is listed first.
In this case the first record written;
PROC SORT DATA=inputds; BY cust id date created; RUN;
*The NODUPKEY keep will keep the first record for that BY group;
PROC SORT DATA=inputds OUT=outds NODUPKEY; BY cust id; RUN;
SAS Code - Number within group;
*Sort records by group;
PROC SORT DATA=inputds; BY cust id; RUN;
DATA outds;
      SET inputds;
      BY cust id;
      IF first.cust id THEN n=1;
      n+1;
RUN;
```

#### **EXAMPLE 13. CREATING TEMPORARY TABLES**

Both SAS and SQL allow users to create temporary tables in memory that are dropped at the end of the user's session. By default, all tables created in SAS that do not have a library specified in the name are temporary tables. SAS stores these tables within the WORK library. Everything in the WORK library is dropped when you end your SAS session. In SQL, a table must include a "#" before the name in order to specify that it is temporary. Another key difference is a user must drop a temp table in SQL before they can recreate it. If the user tries to overwrite an existing temporary table in SQL, the code will error out stating that the table already exists. In SAS, the user does not have to drop the table before recreating. SAS forces the drop and recreates the table.

The single hashtag, "#TMP", syntax tells SQL to remember that table in the current query only. If a double hashtag, "##TMP", is used, SQL will hold the temporary table in memory across all queries open in that session.

```
SQL Code:
IF OBJECT_ID('tempdb..#alldata') IS NOT NULL DROP TABLE #alldata
SELECT *
INTO #alldata
FROM inputds_201411
```

#### SAS Code:

PROC SQL;

SQL Code:

```
CREATE TABLE alldata AS SELECT * FROM inputds_201411; OUIT;
```

### **EXAMPLE 14. CREATING LOCAL/MACRO VARIABLES**

In both SQL and SAS, you have the ability to define a variable value once in your code and reference it throughout your query. These are commonly referred to as local variables in SQL and macro variables in SAS. Most macro variables created in SAS are global and hold the value of that variable in memory as long as your session is open. SQL, however, only allows you to reference the variable while the query is processing. In addition to global macro variables, SAS does have local macro variables which are only available within the macro that defines them. For more information on ways to create macro variables in SAS, see the "Ways of Creating Macro Variables" paper in the recommended reading section.

```
DECLARE @FirstMonday date;
SET @FirstMonday = '1-5-2014';
SELECT *
FROM inputds_201411
WHERE date > @FirstMonday;

SAS Code:
%LET FirstMonday='05jan2014'd;
PROC SOL;
```

PROC SQL;

CREATE TABLE tmp AS

SELECT \*

FROM inputds\_201401

WHERE date > &FirstMonday;

QUIT;

## **CONCLUSION**

This paper outlined many differences we found between SQL and PROC SQL while converting our SQL queries into SAS jobs. This paper is definitely not all encompassing, but is a jump start to understanding why some queries cannot be directly transferred over. As we continue to have our SQL and SAS programmers working closely together, we would like to expand on this topic and provide a more comprehensive paper that includes more differences between the two systems.

We believe the major difference for a SQL programmer to understand while transitioning to SAS, is the way SAS creates datasets. With SQL, many programmers have learned to create table structures then append onto those tables. This is not necessary in SAS. As a SAS programmer, you will spoil yourself with how easy it is to create and override datasets. In our opinion, it is that approach to creating tables that will likely be the biggest thing for programmers transitioning between the languages to get used to.

#### **RECOMMENDED READING**

- PROC SQL for DATA Step Die-hards, Christianna S. Williams, <a href="http://www2.sas.com/proceedings/forum2008/185-2008.pdf">http://www2.sas.com/proceedings/forum2008/185-2008.pdf</a>
- Ways of Creating Macro Variables, Kelley Weston, <a href="http://www.mwsug.org/proceedings/2011/tutorials/MWSUG-2011-TS04.pdf">http://www.mwsug.org/proceedings/2011/tutorials/MWSUG-2011-TS04.pdf</a>
- CAST and CONVERT (Transact-SQL), <a href="https://msdn.microsoft.com/en-us/library/ms187928.aspx">https://msdn.microsoft.com/en-us/library/ms187928.aspx</a>
- Lengths and formats: the long and short of it, Chris Hemedinger, http://blogs.sas.com/content/sasdummy/2007/11/20/lengths-and-formats-the-long-and-short-of-it/

# **CONTACT INFORMATION**

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

Barbara Ross Subprime & Underbanked Analytics Association LLC BMRoss.K@gmail.com

Mary Jessica Bennett Snap Finance LLC Mary.Jessica.Bennett@gmail.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.