Paper 2753-2018

A Transition from SAS® on a PC to SAS on Linux: Dealing with Microsoft Excel and Microsoft Access

Jesse Speer, Ruby Johnson, and Sara Wheeless, RTI International

ABSTRACT

Transitioning from SAS® installed on a PC running Microsoft Windows to a SAS® Grid environment running on Linux presents many challenges for analysts, programmers, and IT professionals alike. Aside from navigating the idiosyncrasies of each operating system, like case sensitivity and the direction of slashes in file pathnames, there arise more significant issues such as handling data from commonly used Microsoft Office products. This discussion focuses on the challenges of importing and exporting data files from Microsoft Excel and Microsoft Access between SAS software on Windows and SAS Grid software (specifically SAS® Enterprise Guide® and SAS® Studio) on Linux. The talk outlines the difficulties encountered and the solutions proposed in order to provide tips for other SAS programmers who must make this type of transition between SAS environments.

INTRODUCTION

Following a growing industry trend, in 2016, RTI International underwent a transition from using individual Windows SAS software licenses to using SAS Grid on Linux. This migration was rolled out over several months with an initial introduction into the essential features of the SAS Grid itself, followed by a tutorial on programming and job-running tips, interspersed with various pilot testing phases to garner initial feedback on unforeseen challenges.

Although the finer details of the SAS Grid are beyond the scope of this paper, it is worth noting that a key difference is that the Grid is a centralized installation of SAS software, while Windows SAS software licenses are specific to the machine on which they are installed. From a programmer's perspective, some of the most important features of the Grid are parallel processing, the ability to access the environment from any physical location connected to the network, and faster execution times (Myers, Allred, and Thissen, 2017).

Although there were some programmers who were familiar with Linux systems, many were inexperienced. Initially, this resulted in a large learning curve, particularly regarding running batch SAS jobs through commands with Linux syntax via the SSH client MobaXterm. A workaround utility was created, running through MobaXterm, to mimic the Windows SAS 9.4 function that batch submits jobs with a simple right-click option. Other challenges included changing file paths to conform to the Linux syntax. One extremely important issue were the changes required to use from Microsoft Office products – specifically Microsoft Excel and Microsoft Access.

This paper addresses issues related to reading data from Microsoft Excel and Microsoft Access in a Linux environment, which is not structured to communicate with Windows programs. It is intended for SAS programmers and statisticians who commonly import data from Microsoft Excel or Access into their programs and must undergo a transition from Windows to Linux.

MICROSOFT EXCEL FILES

SAS has many options available for reading in Microsoft Office files in SAS. Before the transition to SAS Grid on Linux, SAS users at RTI frequently utilized the EXCEL engine or SAS/ACCESS® Interface to PC Files in combination with the EXCELCS engine frequently for this purpose, but adjustments were needed for the SAS Grid environment. The following summarizes the engines and DBMS specifications that can be used for each file type and operating system:

ENGINE BASICS

EXCEL ENGINE

With the EXCEL engine, requiring Microsoft Ace or Jet Provider, SAS can communicate with a driver *of the same bitness* that writes and read any type of Excel file (Schluter and Feldman, 2017). Excel files can be accessed using either the LIBNAME statement or the IMPORT and EXPORT procedures, and all Excel file types are supported. However, since 32-bit Office is often installed and 64-bit SAS is becoming more common, there can be compatibility problems. Further, the same driver is unavailable for UNIX and Linux, so the EXCEL engine is also unavailable. However, one advantage of this engine is that it can be used to read and write older versions of Excel workbooks.

PC FILES SERVER (PCFS) ENGINE

This PCFS engine uses the ODBC interface to communicate with the ACE driver, which allows it to communicate with drivers of any bitness (Schluter and Feldman, 2017). In addition, SAS communicates with the PCFS through a network connection, so it functions on UNIX or Linux. It is important to note here that to call this engine in the LIBNAME statement, you must use the engine name PCFILES, while the DBMS=name for PROC IMPORT and PROC EXPORT is EXCELCS. Like the EXCEL engine, the PCFS engine allows you to handle older versions of Excel workbooks.

XLS ENGINE

The XLS engine does not communicate through a driver, and does not require SAS PC Files Server like the previous two engines. Its biggest advantages are that it reads the format of .xls files directly and can be used on Windows, UNIX, and Linux. However, it can only be used to access Excel 97-2003 files and only in the IMPORT and EXPORT procedures.

XLSX ENGINE

The XLSX engine is like the XLS engine in that they both run without the SAS PC Files Server and are native engines. Unlike the XLS engine, however, it can be used to read Excel 2007 and later files, and has LIBNAME capabilities. Both engines can be used in the IMPORT and EXPORT procedures.

PRE-GRID PC SAS

Before the transition to the Grid on Linux, we often used the EXCEL engine since we did not have a bitness or system compatibility issue and due to its many useful options. Below is some simple example code and Excel workbook typical of our pre-transition method for reading in data:

Table 1.

1							
2	**First Date Event Appeared in Edit**	**ID**	**Payment by Source**			**Records Receipted?**	**Time**
3			**Patient**	**Medicare**	**Medicaid**		
4							
5	2/13/2018	1	$1,000.25	$0.00	$0.00	No	9:00:00 AM
6	2/14/2018	2	$500.00	$200.00	$0.00	No	10:00:00 AM
7	2/15/2018	3	$100.00	$100.00	$50.00	No	11:00:00 AM
8	2/16/2018	4	$0.00	$0.00	$1,000.00	No	12:00:00 AM
9	2/17/2018	5	$300.00	$0.00	$0.00	No	1:00:00 PM
10	2/18/2018	6	$600.00	$0.00	$100.00	No	2:00:00 PM
11	2/19/2018	7	$0.00	$250.00	$0.00	No	11:59:59 PM

Table 1. Example Data (File.xlsx)

Example 1a.

```
libname xls excel "path1\proj\file1.xlsx" access=readonly;
data dates;

      set xls."data$"n(DBSASTYPE=(F1='char(10)' F2=numeric ));

      if F2>0 and F1 ne ' ';

      first_date=input(strip(F1),mmddyy10.);

      id=input(F2,8.);

      if not find(F3,'.') then
      patient=input(cats("$",F3,".00"),dollar8.2);
      else patient=input(cats("$",F3),dollar8.2);

      if not find(F4,'.') then
      medicaid=input(cats("$",F4,".00"),dollar8.2);
      else medicaid=input(cats("$",F4),dollar8.2);

      if not find(F5,'.') then
      medicare=input(cats("$",F5,".00"),dollar8.2);
      else medicare=input(cats("$",F5),dollar8.2);

      if not missing(F6) then time=round(86400*input(F6,15.));

      format patient medicaid medicare dollar8.2;
      format time time8.2;
      format first_date mmddyy10.;

      rename F6=records_receipted_;

run;
libname xls clear;
```

The following is an example that uses PROC IMPORT code to read in the same data:

Example 1b.

```
proc import datafile= " path1\proj\file1.xlsx" out=dates replace
dbms=excel;
      DBDSOPTS= "DBTYPE=(F1='char(8)' F2=numeric)";
      getnames=no;
      range="Data$A2:0";
      mixed=yes;
      usedate=yes;
      scantext=yes;
      scantime=yes;
      textsize=32767;
run;
```

As detailed above, the EXCEL engine specified in the LIBNAME statement and in the DBMS parameter is not available in the Linux environment. Since we are dealing with a newer Excel file (the .xlsx file ending indicates the file is Excel 2007 or later), we can use either the XLSX or PCFS engines. In our transition, we no longer had access to the PC Files Server Excel Driver, so we defaulted to the XLSX engine. Of the options shown in the PROC IMPORT step above, only GETNAMES= and RANGE= are still available in the XLSX engine. In addition, there were other adjustments required behind the scenes before reading in this data, but these differ based on the method chosen.

GRID LINUX SAS

There are several key differences that apply between all the pre-Grid-Windows code examples and the post-Grid-Linux code examples:

1. The direction of slashes in the pathname changes from '\' to '/' in the Linux environment.
2. The server location (the beginning of the path) also changes as files are now connected to the Grid. In most cases, this means pathnames have lengthened.
3. Linux is case-sensitive while Windows is not. In Example 1a/1b above, the file was read in successfully even though its true name is File1.xlsx (not file1.xlsx).

There are several approaches to transitioning the Excel code above. Four approaches are described in this section:

1. LIBNAME and Data Step with no manual changes to the file before importation
2. LIBNAME and Data Step with manual changes to the file before importation
3. The Import Procedure
4. The INFILE statement

LIBNAME and Data Step – No File Changes

If there are few variables of interest, as in the example above, it may make sense to keep the code relatively similar as shown below:

Example 2a.

```
options validvarname=v7;
libname xls xlsx "/path1/path2/proj/File1.xlsx" access=readonly;
data dates;
     set xls.Data (firstobs=2);

     informat A $10. ;
     informat B 4. ;
     informat C $10. ;
     informat D $10. ;
     informat E $10. ;
     informat F $4. ;
     informat G $15. ;

     if not missing(A) and not missing(B);

     if not missing(A) then first_date=input(strip(A),10.)-21916;

     id=input(B,8.);

     if not find(C,'.') and not missing(C) then
     patient=input(cats("$",C,".00"),dollar8.2);
     else patient=input(cats("$",C),dollar8.2);
     if not find(D,'.') and not missing(D) then
     medicaid=input(cats("$",D,".00"),dollar8.2);
```

```
      else medicaid=input(cats("$",D),dollar8.2);
      if not find(E,'.') and not missing(E) then
      medicare=input(cats("$",E,".00"),dollar8.2);
      else medicare=input(cats("$",E),dollar8.2);

      if not missing(G) then time=round(86400*input(G,15.));

      format first_date mmddyy10.;
      format patient medicaid medicare dollar8.2;
      format time time8.2;

      rename F=records_receipted;

      drop A B C D E G H I J K;

run;
```

Here you can use the FIRSTOBS=option to ignore the first row of blank cells from your workbook and begin reading on the second row. However, this still reads the default names of variables (A, B, C, etc.) rather than assigning the fields of the second row as variable names. In addition, the DATA step guesses where the data ends, so you need to use a subsetting IF statement to delete rows without essential data, like the blank row between the subheader variable names and the beginning of the data. Also, the DATA step will sometimes read additional unnecessary columns (H, I, J, K), but these can be dropped in the DROP statement.

This option also makes apparent an issue with reading in dates when they have a general format in Excel, in that the software reads them as the character version of the SAS date number (the number of days since January 1, 1960), but with 60 years added to the original date. In addition, some manipulation may be required when reading in time data. When importing, if SAS does not recognize the column as a time variable, as in this example, it will read in the values as a number between 0 and 1, with 0 corresponding to midnight and 1 corresponding to the nanosecond of time just before midnight. For its own TIME format, SAS requires a number (numerically formatted) to be between 0 and 86,400, where 0 corresponds to midnight and 86,400 is the time just before midnight. Dates are shown in column A and times in column G below:

Table 2

A	B	C	D	E	records_receipted	G
43144	1	1000.25	0	0	No	.375
43145	2	500	200	0	No	.41666666666666674
43146	3	100	100	50	No	.45833333333333326
43147	4	0	0	1000	No	0
43148	5	300	0	0	No	.5416666666666666
43149	6	600	0	100	No	.5833333333333334
43150	7	0	250	0	No	.999988425925926

Table 2. File1 After Import (No Manipulation)

To remedy the strange dates, you must subtract 60 years in days (21,916) from the numeric version of column A, and then format that numeric result with your date format of choice, which is MMDDYY10. For a dataset with many date columns, this option may be infeasible. To get the times in a sensible format, you can multiply their numeric versions by 86,400 and then format the result using the TIME format.

Here, as with the rest of the examples that follow, the DBSASTYPE option from SAS/ACCESS that allowed us to convert variables to the SAS data type of our choosing upon input processing is unavailable. However, this option did have its limitations. Because Excel allows each cell to have its own data type, it is possible to have columns with a mix of numeric and character values, unlike in SAS software, which requires uniformity in a variable. The MIXED(=yes) option, available in the EXCEL engine, was the solution to this issue, but would fail in a situation where the first 8 rows of a mixed type column contained numeric values (unless the user changed the SAS registry). Even with the addition of DBSASTYPE, the problem remained because the EXCEL engine already determined the type within reading the first 8 rows of data by default. The XLSX engine, on the other hand, uses all rows, and converts to character any column that contains at least one character cell. In this case, SAS reads columns C, D, and E, which contain a subheader variable name along with currency values, as character.

To gain more control of how variables are read in, you can use a combination of INFORMAT and PUT/INPUT functions to convert your variables to the correct type, assuming they are not read in correctly the first time. Additionally, the XLSX engine, unlike the EXCEL engine, does not require the use of the name literal to refer to specific sheets in the Excel workbook. Thus, you can now read in sheets simply by placing their names after the libref (no longer requiring quotation marks, “$”, and an “n” following the sheet name).

LIBNAME and Data Step – File Changes

Assuming a small number of Excel files, a simple solution may be to delete rows at the top of the Excel workbook to force variable names to the first row of the file. Since Excel does not have the same variable conventions as SAS, this solution especially requires that you use the validvarname=v7 option; this will ensure that all variables read in with spaces are replaced with underscores and other naming rules are followed. The data after manual cleaning would be as follows:

Table 3

First Date	ID	Patient	Medicare	Medicaid	Records Receipted?	Time
2/13/2018	1	$1,000.25	$0.00	$0.00	No	9:00:00 AM
2/14/2018	2	$500.00	$200.00	$0.00	No	10:00:00 AM
2/15/2018	3	$100.00	$100.00	$50.00	No	11:00:00 AM
2/16/2018	4	$0.00	$0.00	$1,000.00	No	12:00:00 AM
2/17/2018	5	$300.00	$0.00	$0.00	No	1:00:00 PM
2/18/2018	6	$600.00	$0.00	$100.00	No	2:00:00 PM
2/19/2018	7	$0.00	$250.00	$0.00	No	11:59:59 PM

Table 3. File1.xlsx After Deleting Rows

The code is adjusted below:

Example 2b.

```
options validvarname=v7;
libname xls xlsx "/path1/path2/proj/File1_edited.xlsx"
access=readonly;
data dates ;
   set xls.Data;

   if not missing(id) and not missing(first_date);
   drop H I J K;
run;
libname xls clear;
```

SAS still does some guesswork as to where the data ends, so you still need to keep the subsetting IF statement and DROP statement to ensure no blank rows or columns are kept. Other than that, there is very simple code here since most of the re-arranging occurred on the front end. In this case, all the variables were recognized as the correct format the first time, and we renamed before reading the data in, so none of that manipulation is required.

This option highlights the strength of the XLSX engine compared with the EXCEL engine. Whereas with the EXCEL engine, we needed the MIXED= option to tell SAS to read mixed type columns as character, the engine now does this automatically. In addition, the XLSX engine does not need USEDATE= and SCANTIME= options to correctly read and assign TIME. and DATE. formats. In fact, the USEDATE= option was limited by the fact that it only assigned columns a DATE9. format whereas XLSX reads our data in its original MMDDYY10. format in this case. Moreover, the SCANTEXT= and TEXTSIZE= options, which told SAS whether to scan text columns for variable length and then what that maximum length should be, respectively, are no longer necessary as SAS automatically scans the whole text column and assigns the length of the longest text/character string. SCANTEXT= and TEXTSIZE=, however, would only read all rows for variable length if the SAS registry settings are edited.

The only downside to this importation option is that it may not be efficient for many Excel files or if programs are run on an automated, periodic basis (e.g. a job that runs nightly).

Import Procedure

Another simple, but somewhat more dynamic solution is to use PROC IMPORT with the RANGE= option, which will indicate the entire range to be read, including the location of the variable names. The RANGE= option ends the need for options STARTCOL=, STARTROW=, ENDCOL=, and ENDROW= (from the XLS engine), which as their names imply, instructed SAS as to which rows and columns to begin and end reading from the Excel workbook. The code presented in the first example below would not be suitable for many Excel workbooks or for an Excel workbook that is frequently updated with additional columns, or more likely rows. The code would be as follows:

Example 2c1.

```
options validvarname=v7;
PROC IMPORT OUT=dates FILE="/path1/path2/proj/File1.xlsx" dbms=xlsx
REPLACE;
   range="Data$A2:G11";
RUN;
```

Although options with the XLSX engine are limited, you only need the RANGE= option in this case. By specifying the procedure to start reading in the second row (the “2” in “A2”), you also instruct it to read variable names from here. In this way, you do not need the GETNAMES= or NAMEROW= options, which tell the procedure from whence to generate column names. You can also feed the name of the sheet using the RANGE= option (the sheet is called “Data” in this case), so you do not need the SHEET= option. To make this method more dynamic, we can amend the code as follows:

Example 2c2.

```
options validvarname=v7;
PROC IMPORT OUT=dates FILE="/path1/path2/proj/File1.xlsx" dbms=xlsx
REPLACE;
   range="Data$A2:0";
RUN;
```

As an aside, PROC IMPORT is the only option that seems to work when reading an Excel Macro-Enabled Workbook (.xlsm file), but takes a slight modification as shown below:

Example 2c3.

```
%let file='/path1/path2/proj/File1.xlsm’ access=readonly;
options validvarname=v7;
PROC IMPORT OUT=dates FILE="&file." dbms=xlsx REPLACE;
   range="Data$A2:0";
  RUN;
```

When trying to read the workbook directly in PROC IMPORT, SAS will try to read a file called “File1.xlsm.xlsx”, which of course does not exist. However, if you place the file name in single quotes with the option ACCESS=readonly into a macro variable, the file will be read correctly, just as in the above two examples. The DATA step that follows would be the same whether you began with a Macro-Enabled Workbook or not:

```
data dates2;
set dates(rename=(first_date_event_appeared_in_edi=first_date));

if id>0 and first_date ne .;

if not find(payment_by_source,'.') then
patient=input(cats("$",payment_by_source,".00"),dollar8.2);
else patient=input(cats("$",payment_by_source),dollar8.2);
if not find(D,'.') then medicaid=input(cats("$",D,".00"),dollar8.2);
else medicaid=input(cats("$",D),dollar8.2);
if not find(E,'.') then medicare=input(cats("$",E,".00"),dollar8.2);
else medicare=input(cats("$",E),dollar8.2);
format patient medicaid medicare dollar8.2;

drop payment_by_source D E H I J K;
run;
```

In some cases, Examples 2c2 and 2c3 will lead to SAS reading in more rows and columns than necessary. This is because the “0” in the RANGE= statement tells SAS to determine the last row and column. However, if your data are updated frequently, and it is unreasonable to constantly update the range, this method is a solution, especially in combination with the subsetting IF statement. PROC IMPORT will still begin reading from the second row and thus read variable names correctly. But since there are columns with variable names in the wrong row, you still need to rename and manipulate to ensure the data is read and displayed in the correct format:

Because the subheader variable names were read within the same columns as the currency values, the XLSX engine forced these variables to be typed as character (the XLSX engine assumes MIXED=yes). You can essentially reapply the format to these columns and use the INPUT function to ensure that these are correctly converted to numeric values. Then you can apply the proper format with the FORMAT statement. We did not undergo this process with the first or last columns, *first_date_event_appeared_in_edi* and *time*, because they did not contain variable names in the second row. In other words, since all their values were of the same type, the procedure read them correctly as dates and times immediately.

INFILE Statement

Another solution – again assuming a small number of files and variables, but also a placement of variables all in one row – is to convert the file to comma-separated values (csv) format. When the original data is in Excel, you can do this simply by changing the file type to “.csv” when using the “Save As” feature. Although the new file can still be opened with Excel and viewed as a workbook, its structure has been reduced to a much simpler text file format that can be read by text editors such as Notepad and WordPad. In its true form, after converting to .csv and manually moving variable names to one row, the new data would look like this:

```
Table 4

,,,,,,,,,

First Date Event Appeared in Edit,ID,Patient,Medicare,Medicaid,Records Receipted?,Time,,,,

,,,,,,,,,

2/13/2018,1,"$1,000.25",$0.00,$0.00,No,9:00:00 AM,,,,

2/14/2018,2,$500.00,$200.00,$0.00,No,10:00:00 AM,,,,

2/15/2018,3,$100.00,$100.00,$50.00,No,11:00:00 AM,,,,
```

Table 4. File1 in Raw Csv Format

This simpler format means the file no longer contains information about variable types and lengths, so they must be provided upon importation. The downside is, of course, that this might involve a large effort on your part, depending on the number of variables. On the other hand, this gives you total control of how the data is imported and does not leave any room for guesswork by SAS. Thus, this cuts down on the number of unexpected results. A code example is shown below:

Example 2d.

```
data dates;

infile "/path1/path2/proj/file1.csv"
                 delimiter=','
                 missover
                 firstobs=3
                 DSD
                 lrecl = 32767;

    informat first_date mmddyy10.;
    informat id $4.;
    informat patient dollar8.2;
    informat medicare dollar8.2;
    informat medicaid dollar8.2;
    informat records_receipted $3.;
    informat time time8.2;

    format first_date mmddyy10.;
    format patient dollar8.2;
    format medicare dollar8.2;
    format medicaid dollar8.2;
    format time time8.2;

    input     first_date
              id $
              patient
              medicare
              medicaid
              records_receipted $
              time;
 run;
```

In the INFILE statement, you must provide the following:

1. The location of the CSV file - "path1/path2/file1.csv" in this example.
2. Delimiter option – the delimiter, or separator between fields, which in this case is a comma.
3. MISSOVER option – This instructs SAS to read rows all the way through even if they contain missing values.
4. FIRSTOBS – The first row with data in the input file, which is set to 3 here. Note that we have variable names in row 2, but we input them ourselves using the INFORMAT, FORMAT, and INPUT statements.
5. DSD –Causes SAS to read two consecutive delimiters as a missing value (two commas in a row) and removes quotation marks from character values.
6. LRECL – Logical record length, or the maximum length for a record (32,767 is the maximum number allowed here).

Also note that here you assign your own variable names, as long as they conform to SAS variable name standards and conventions. In the .csv format, you must ensure that you assign variable names to each column in the INPUT statement in the order in which they appear in the file.

MICROSOFT ACCESS FILES

PRE-GRID PC SAS

Although Microsoft Access files are less common, we still receive a fair amount of data in this format. There are less engines available to read these files, but their functions and options are very similar to those of the Excel engines. On Windows PC SAS, you can use the ACCESS engine to read an Access file from any version beyond Access 97 (for which the engine ACCESS97 would be used). A simple code example follows:

Example 3.
```
proc import out=table
            datatable= "table1"
            dbms=access replace;
                database="\\path1\proj\File2.accdb";
               scanmemo=yes;
               usedate=no;
               scantime=yes;
run;
```

To analogize to Excel, the DATATABLE= statement is like the SHEET= statement just as sheets of an Excel workbook are analogous to tables in an Access database. Similarly, the DATABASE= statement for importing Access is like the DATAFILE= statement for importing Excel in that this is where you provide the file name and location. The option SCANMEMO= above basically performs the same function as SCANTEXT= in PROC IMPORT with the EXCEL engine in that it instructs SAS to use the longest MEMO data field to determine the column length for MEMO data type columns. As with the EXCEL engine, to avoid truncation, you would need to modify SAS registry options to override the default number of rows used by the driver to determine the variable length. The options USEDATE= and SCANTIME= perform the exact same function as they did in the EXCEL engine. When USEDATE=no, as above, the DATETIME. rather than the DATE9. format is applied to date/time columns. The code is currently set up to correctly read and format time values.

GRID LINUX SAS

We did not have access to the SAS PC Files server upon transitioning to Linux Grid SAS (which would have allowed us to replace the DBMS=ACCESS statement with DBMS=ACCESSCS). As an alternative, we use an ODBD-ACCESS driver created by Easysoft Limited, which can read and write MS Access files on Linux through the SAS/ACCESS Interface to ODBC engine. The following code uses the LIBNAME statement with the Easysoft driver:

Example 4a.

```
   libname tab ODBC complete="DRIVER=Easysoft ODBC-ACCESS;
   dbq=/path1/path2/proj/File2.accdb;
   readonly=yes;
   exclusive=no;
   ignore_rel=no;
   work_dir_path=/saswork/grid1;
   blob_path=/saswork/grid1;";

  data table;
    set tab.Table1;
  run;
```

In the LIBNAME statement, first you specify the libref (“tab”), followed by the engine name (“ODBC”) and then use the COMPLETE= option to provide SAS with the connection options for the Access data. Each option is separated by a semicolon as shown above. Note that all the lines need to be flush (or all on one row) for the code to execute properly.

Firstly, we specify the name of the driver in the DRIVER= option, followed by the location to the file in the DBQ= option. Then the READONLY= option indicates whether the connection should be able to write to the database. Similarly, the EXCLUSIVE= option indicates whether the database should be locked, which prevents other users from accessing it while the process is connected to it. The option IGNORE_REL= indicates whether the driver considers any relationships defined for a particular table when doing INSERTs, UPDATEs or DELETEs. The two final options, WORK_DIR_PATH = and BLOB_PATH=, provide the location where the driver temporarily stores results set rows when the memory cache is exceeded. Equivalently, you can use the following code to read the file with the SQL procedure:

Example 4b.

```
proc sql;
      connect to odbc as table (complete="Driver=Easysoft ODBC-ACCESS;
      dbq=/path1/path2/proj/File2.accdb;
      readonly=yes;
      exclusive=no;
      ignore_rel=no;
      work_dir_path=/saswork/grid1;
      blob_path=/saswork/grid1;");
quit;
```

The main difference here is the syntax to connect to the ODBC engine, using the CONNECT statement to specify the DBMS to which you want to connect. The options specified in the COMPLETE= option are exactly the same.

CONCLUSION

While the transition between Windows PC SAS and SAS Grid on Linux may at first seem overwhelming, with experience and knowledge of the differences that truly make a difference in your programs, you may begin to see the operating system differences as mere quirks rather than frustrating roadblocks. Additionally, once you have sorted out basics like how batch jobs may run differently, most of the differences between PC SAS and SAS Grid will be behind the scenes and will likely mean improved processing speed for you. Nevertheless, it can be frustrating when programs you have been running successfully for years begin to crash, and suddenly options that you have always depended on to access data disappear. However, as demonstrated here, many of the options specific to Excel and Access engines for Windows are either replicated in engines available for Linux or equivalent workarounds can be undertaken. Only if you are regularly reading and writing older versions of Excel files will you miss the

SAS/ACCESS to PC Files server. Otherwise, you will likely find your code simplified and your programs running more efficiently.

REFERENCES

Easysoft. “Easysoft ODBC-Access Driver User's Guide - Configuration.” (n.d.). Available at https://www.easysoft.com/products/data_access/odbc-access-driver/manual/configuration.html#880541.

Hemedinger, C. (May 20, 2015). “Using LIBNAME XLSX to Read and Write Excel Files”. The SAS Dummy. Available at https://blogs.sas.com/content/sasdummy/2015/05/20/using-libname-xlsx-to-read-and-write-excel-files/#prettyPhoto.

“Importing CSV Files into SAS.” SASCrunch. Available at https://www.sascrunch.com/importing-csv-files.html.

Myers, S., Allred, I., and Thissen, M. R. (2017). “Migrating from PC SAS to SAS Grid on Linux.” In Proceedings of the SouthEast SAS Users Group (SESUG) 2017 Conference. Cary, NC. Available at http://www.sesug.org/SESUG2017/.

Rabb, M., and Brown, K. (2017). “Parallel Processing in a SAS Grid.” SouthEast SAS Users Group (SESUG) 2017 Conference, Cary, NC. Available at http://www.sesug.org/SESUG2017/.

SAS Institute Inc 2010. SAS® 9.2 for Relational Databases: Reference, Fourth Edition, Cary, NC: SAS Institute Inc.

SAS Institute Inc 2010. SAS® 9.2 Language Reference: Dictionary, Fourth Edition, Cary, NC: SAS Institute Inc.

SAS Institute Inc 2017. SAS/ACCESS® 9.4 Interface to PC Files: Reference, Fourth Edition, Cary, NC: SAS Institute Inc.

SAS Institute Inc 2017. SAS/ACCESS® 9.4 Interface to PC Files: Reference, Fourth Edition, Cary, NC: SAS Institute Inc.

SAS Institute Inc 2017. SAS® 9.4 DATA Step Statements: Reference, Cary, NC: SAS Institute Inc.

Schluter, J., and Feldman, H. (2017). “SAS/ACCESS® Interface to PC Files: So Many Options for Microsoft Excel Files — Which Is Best for Me?” SAS Global Forum 2017, Orlando, FL. Available at https://support.sas.com/resources/papers/proceedings17/SAS0387-2017.pdf.

CONTACT INFORMATION

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

Jesse Speer
RTI International
jspeer@rti.org