Paper 001-2007

How to Generate 10,000 Excel Spreadsheets in 10 Minutes (Or Less)

Alan Churchill, Savian, Colorado Springs, CO

ABSTRACT

SAS programmers have traditionally used techniques such as proc export, DDE, Access to PC File Formats, excel libname engine, and ODS to generate Excel spreadsheets. Each one of these approaches, though, focuses on 'pushing' the data (Excel spreadsheets) from SAS.

A new .NET 'pull' technique is presented in this paper. This technique 'pulls' the data from SAS and creates Excel spreadsheets from a dedicated Excel model. This approach allows for full customization of Excel spreadsheets including formatting, calculations, printing styles, etc. using SAS datasets. In addition, native Excel spreadsheets are generated. This approach, by being a native .NET application, is an amazingly fast way of generating Excel spreadsheet and it overcomes many of the limitations seen in the past.

INTRODUCTION

SAS programmers should be familiar with proc export, DDE, ODS, et al to create Excel spreadsheets: these are the tools of the trade and numerous examples exist for how to use these techniques. Each one of these techniques relies on a 'push' operation whereby SAS pushes the data into Excel. However, each one of these techniques has various shortcomings. These limitations range from using the Excel COM model (which only allows 1 instance of Excel at a time) to not writing worksheets in Excel's native format. While they work for many applications, these issues can cause significant problems under the right circumstances.

Recently, a large merger (1000+ stores) took place. This merger created a need for massive amounts of information. How do you tell which store has what inventory? How much did they sell the previous year? Who is their companion store? What are their vendors? etc. Statisticians generated a copious amount of data using SAS that needed to be seen and acted upon immediately since there was a short timeframe for completing the merger. All of the data was in SAS datasets but the PCs in every store had 1 thing to rely on for information delivery and management: Excel.

The requirement in this case was to generate multiple spreadsheets for every store in the nation, every department, every vendor….and then do this multiple times per day. The Excel spreadsheets needed to be tweaked continually to get the formats right, change layouts, etc. Borders, formulas, fonts, shading, cell level writing, print layout…all were needed depending upon the sheet in question and the analysis data presented. All in all, thousands of spreadsheets per run with multiple runs per day.

This need required a new way of thinking about the process that a traditional approach would take. Instead of doing a 'push' technique, the new process would use a 'pull' technique to allow for complete customization. Initially, the approach was to pull the data using Excel's COM in a C# program. However, this meant that only 1 instance of Excel can run per machine. After ramping up 6 PCs to do the processing, it was still taking 12+ hours a day to generate all of the reports. A mistake in a report meant reprocessing and further delays.

Ultimately, the solution devised was to use an Excel object model that was .NET based. This proved to be not only easy to code against but also very, very fast since no COM interop was involved. This solution was able to generate fully formatted Excel spreadsheets at an astounding rate of 50 per second (actual results varied depending upon the format options and the data in a given sheet).

[Note: The code shown in this paper is C#. C# is a newer language and is supported on all current versions of Windows through the .NET Framework.]

OVERALL PROCESS

SUMMARIZATION

The process for creating the spreadsheets relies on reading the SAS datasets using OleDb, which SAS supports using a free add-in. The process also relies on a 3rd party product called Aspose Cells which is a .NET native library for generating Excel spreadsheets.

Here is the overall flow of the process:

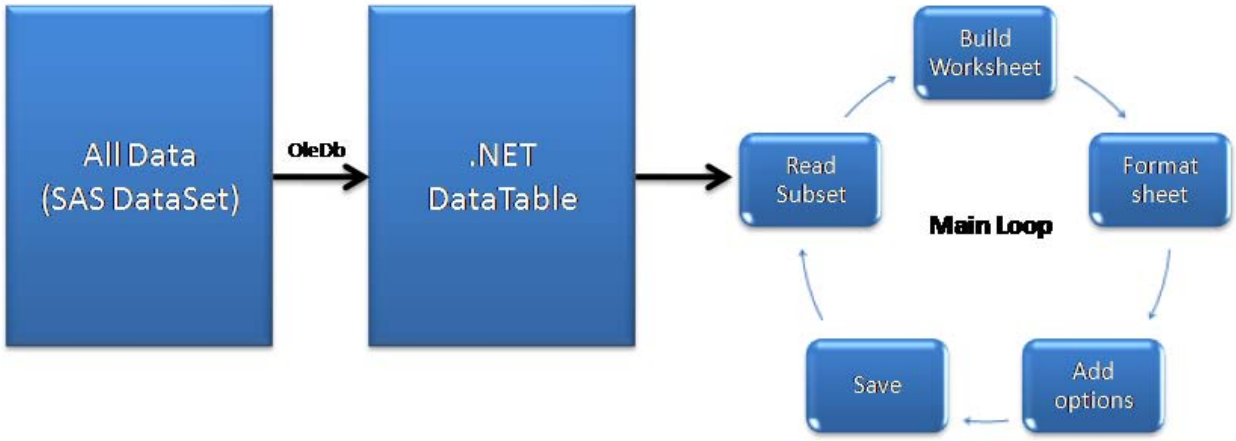


Figure 1 Overall Process

Data is read from SAS using OleDb and then stored into an in-memory .NET datatable. Then a loop runs over the data pulling out subsets as needed and creating the worksheets.

There are some key advantages to this approach:

- Speed.
- Full random access to any cell within the sheet at anytime. This greatly simplifies logic and programming
- Easy to use Excel object model.
- Follows Excel model where it makes sense
- Low cost 3rd party product (~$400)
- Does not rely on SAS software (except OleDb client). This means that the application can run on non-SAS systems.
- Can operate on more than 1 Excel instance at a time
- Does not open Excel or require it on the PC
- Generates native Excel spreadsheets

OLEDB

SAS datasets are OleDB compliant and therefore can be read by numerous tools and technologies. While many users are familiar with ODBC, fewer use OleDB even though it is a newer technology. However, OleDB is easier to work with, is more standards-oriented, and can be easily used in coding languages such as C#, Java, Perl, etc.

SAS provides a free, client-side tool that allows for a dataset to be read using OleDB. Simply run the setup for the SAS OleDB driver and then call SAS using OleDB.

```
class SasManagement
{
    /// <summary>
    /// Loads a SAS dataset into a DataTable
    /// </summary>
    /// <param name="sasLibrary">The physical path to a SAS
    ///                          library</param>
    /// <example>C:\temp\SasLibrary</example>
    /// <param name="sasDataSet">The name of the sas dataset</param>
    /// <returns></returns>
    public DataTable LoadSasDataSet(string sasLibrary, string sasDataSet)
    {
        DataTable dt = new DataTable();
        OleDbConnection sas = new OleDbConnection("Provider=sas.LocalProvider; Data Source=" + sasLibrary);
        sas.Open();
        OleDbCommand sasCommand = sas.CreateCommand();
        sasCommand.CommandType = CommandType.TableDirect;
        sasCommand.CommandText = sasDataSet;
        OleDbDataReader sasRead = sasCommand.ExecuteReader();
        dt.Load(sasRead);
        endTime = DateTime.Now;
        return dt;
    }
}
```

In the above code, we see a method call named LoadSasDataSet. It takes in 2 strings, sasLibrary and sasDataSet, and returns a .NET datatable (a .NET datatable is akin to a SAS dataset). An instance of a OleDb data connection is created, opened, and the specified SAS dataset is read into the datatable instance.

This method is easily called as:

```
SasManagement sas = new SasManagement();
dt = sas.LoadSasDataSet(@"C:\Program Files\SAS\SAS
                        9.1\core\sashelp", "shoes");
```

Resulting in a new datatable called dt.

ASPOSE AND EXCEL

Excel comes with a built-in object model that relies on the pre-.NET COM technology. Originally, this was the technology that was employed to create all of the spreadsheets. However, that technology had numerous limitations such as application locking, speed, older object model, difficult to code and debug, etc. Using that model, it took 6 PCs a half-day each to run the generation. It worked but you could only call 1 instance of Excel per PC because COM would step on any other instances.

To solve this problem, a decision was made to switch to a .NET based Excel object model made by a company called Aspose. Known as Aspose Cells, this object model is written in .NET and suffers none of the problems seen by COM. It can run multiple instances, runs very, very fast, is easy to set up, and is very easy to code and debug.

Here is some example code (adjusting print settings) showing the COM model exposed in .NET:

```
xlSheet.PageSetup.PrintTitleRows = "R1C1:R7C26";
xlSheet.PageSetup.FitToPagesWide = 1;
xlSheet.PageSetup.FitToPagesTall = false;
```

Here is similar code in Aspose:

```
xlSheet.PageSetup.PrintTitleRows = "R1C1:R7C26" ;
xlSheet.PageSetup.FitToPagesWide = 1;
xlSheet.PageSetup.FitToPagesTall = 1000;
```

They are virtually identical. The Aspose model was a good choice as well since the move minimized the amount of recoding necessary.

The Aspose model was a true lifesaver and it should be used for worksheet creation where possible. The remainder of the paper will use code examples from Aspose. If a desire is there to use the Excel object model instead then the code shown should be very close to what would be used and can easily be translated to the COM model.

BUILD A WORKSHEET

Creating a worksheet is fairly straightforward:

```
Aspose.Excel app = new Aspose.Excel() ;
Aspose.Excel.Worksheet sheet= app.Worksheets.Add() ;
sheet.Name = “Test Sheet” ;
```

Once the sheet is created then the process of modification can start. Here, a new cell object is created, the cell is merged, and a new value is placed inside the cell:

```
int row = 5 ;
Cells cells = sheet.Cells;
cells.Merge(row - 1, 1, 1, 21);
cells[row - 1, 1].PutValue("FY 2004");
```

Styles can be easily applied to a cell as well:

```
Aspose.Excel.Style style = app.Styles.Add();
style.Name = “Header 1”;
style.HorizontalAlignment = TextAlignmentType.Left;
style.Font.Name = "Tahoma";
style.Font.Size = 8;
style.Font.IsBold = true;
style.VerticalAlignment = TextAlignmentType.Top;
style.ForegroundColor = Color.Red;
```

…as can printing options:

```
sheet.PageSetup.Orientation = PageOrientationType.Landscape;
```

```
sheet.PageSetup.SetHeader(0, “Title of Report”);
sheet.PageSetup.SetFooter(0,"My Company Name");
sheet.PageSetup.SetFooter(2,"Confidential information.");
sheet.PageSetup.SetFooter(1,@"Page: &P");
```

When the report is completed, simply save the new Excel file in native Excel format:

```
app.Save(@”c:\temp\MyFile.xls”) ;
```

WORKING WITH SAS DATA IN THE MODEL

Since the SAS dataset is now in a .NET datatable, it is easy to pull together the new SAS data and put it into our Aspose Excel model.

Create a Cells object:

```
Cells cells;
```

Then create the worksheets:

```
private void AddWorksheet(int j)
{
    sheet.Name = "SAS Global Forum Demo " + j;
    cells = sheet.Cells;
    sheets++;
    AddStyles();
    AddHeader();
    int i = AddData();
    AddFormulas(i);
}
```

Create the header row:

```
private void AddHeader()
{
    cells[0, 0].PutValue("Test Data");
    cells[0, 0].Style = app.Styles["Blue"];
    cells[2, 0].PutValue("Region");
    cells[2, 1].PutValue("Product");
    cells[2, 2].PutValue("Subsidiary");
    cells[2, 3].PutValue("Stores");
    cells[2, 4].PutValue("Sales");
    cells[2, 5].PutValue("Inventory");
    cells[2, 6].PutValue("Returns");
}
```

Then add the data:

```
private int AddData()
{
    int i = 3;
    foreach (DataRow dr in dt.Rows)
    {
        cells[i, 0].PutValue(dr["Region"]);
        cells[i, 1].PutValue(dr["Product"]);
        cells[i, 2].PutValue(dr["Subsidiary"]);
        cells[i, 3].PutValue(dr["Stores"]);
```

```
            cells[i, 4].PutValue(dr["Sales"]);
            cells[i, 5].PutValue(dr["Inventory"]);
            cells[i, 6].PutValue(dr["Returns"]);
            i++;
        }
        return i;
    }
```

The above is fairly simple code that can easily be expanded using the techniques shown above. In this sample code, the SAS data is looped through and the columns in the .NET datatable are written to the Excel spreadsheet. The row counter, i, is then incremented and the next row is written.

Finally, add in the equations:

```
    private void AddFormulas(int i)
    {
        cells.CreateRange("d" + i, "g" + i).SetOutlineBorder(BorderType.BottomBorder, CellBorderType.Thick,
Color.Black);
        cells[i, 3].Formula = "=SUM(d3:d" + i + ")";
        cells[i, 3].Style = app.Styles["Commas"];
        cells[i, 4].Formula = "=SUM(e3:e" + i + ")";
        cells[i, 4].Style = app.Styles["Commas"];
        cells[i, 5].Formula = "=SUM(f3:f" + i + ")";
        cells[i, 5].Style = app.Styles["Commas"];
        cells[i, 6].Formula = "=SUM(g3:g" + i + ")";
        cells[i, 6].Style = app.Styles["Commas"];
    }
```

Complete control over the spreadsheets is afforded and the object model makes sense and flows similar to DDE or a VBA approach.

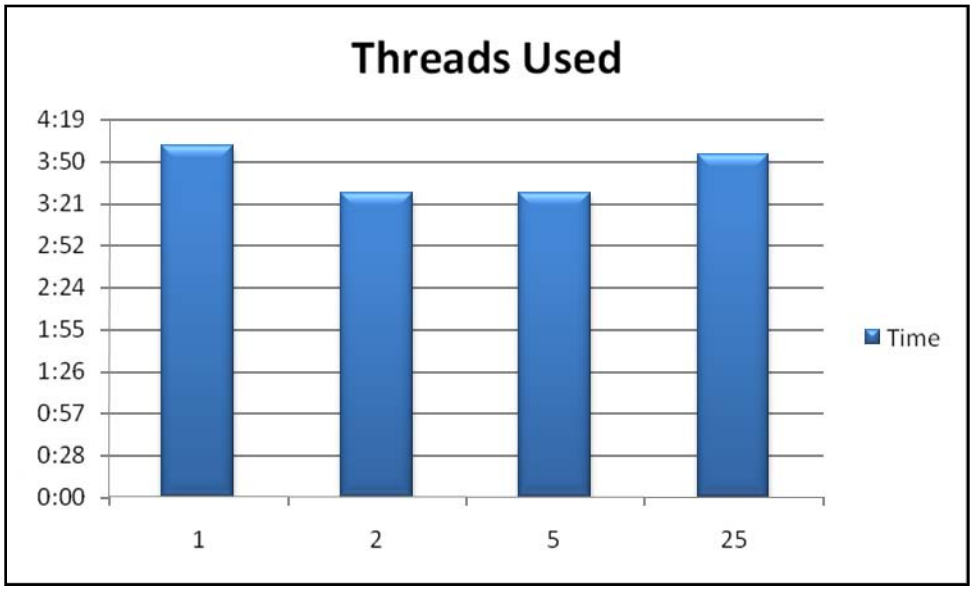
PERFORMANCE

OVERALL

Using the above techniques, the sashelp.shoes dataset is written to the Excel sheets at approximately 0.04 seconds/sheet (10,000 sheets, 6.5 minutes). This is all running on a Toshiba laptop with a decent processor (2Ghz). The question then becomes, can it go faster?

THREADING

Microsoft's .NET comes with a native threading engine. After working with this threading engine and trying it at various levels, threading does not increase the overall speed. Here are the results:



As is evident, adding additional threads does not really improve performance. But where are the bottlenecks here? If the actual file write to disk is turned off, the results are almost the same but a minute is saved overall across the board. Hence, it doesn't appear that I/O is the culprit. Where the issue seems to lie is with CPU which consistently runs at 90+% regardless of the threads employed. A multi-core CPU should then reduce the overall time even further, hopefully cutting it in half and getting to the elusive 10,000 worksheets in 1 minute.

CONCLUSION

The method shown in this paper has a number of advantages over traditional methods used to move SAS data into Excel. By changing the focus to an Excel model driven process, the full power of Excel becomes available. Using the above technique allows for very reusable code, is the fastest method known for generating Excel spreadsheets, generates real Excel files, and the process easily reads in SAS datasets.

While the challenges faced by the merger were of a large scale, they ultimately led to a very fast and efficient solution for SAS to Excel processing. Going from processes that lasted all day across 6 machines to a process that occurs in mere minutes on 1 machine was a solid achievement.

Many SAS programmers may be reluctant to switch from a push methodology to a pull methodology but it truly illustrates the advantages of using the power of Excel to manage Excel.

CONTACT INFORMATION

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

Alan Churchill
Savian
68 W Cheyenne Mtn. Blvd
Colorado Springs, CO 80906

Work Phone: 719-687-5954
E-mail: alan.churchill@savian.net
Web: www.savian.net