Best Practices with Large Reports

Introduction

ExcelWriter is a powerful tool for generating and manipulating Excel files. Files can be saved to disk or streamed to the user over HTTP. In order to provide maximum flexibility for manipulating every element of a spreadsheet, ExcelWriter (in particular ExcelApplication) has a rich object model that must be populated at runtime and requires sufficient memory.
It is important to understand that the memory required to process a large report is much greater than the size of eventual output file.

Any cell loaded by the ExcelApplication object model may require up to 400 bytes of memory for the various objects associated with each cell for values, formulas, formatting, etc. The number of cells equals the number of rows times the number of columns. So if you have a large report with 50,000 rows by 20 columns, that's 1 million cells and it can use up 300-400 MB of memory to process just one request for that report.

Here is a table showing approximately how much memory is required for large reports with different column/row combinations:

rows	columns	cells	memory (bytes)	memory (MB)
1,000	10	10,000	4,000,000	4
20,000	20	400,000	160,000,000	153
50,000	20	1,000,000	400,000,000	381
50,000	30	1,500,000	600,000,000	572

If you have a complex workbook with many sheets and you aren't sure how large it is, here's a simple macro you can use for calculating the total number of cells and approximate memory required:

Private Sub CalculateCells

```
Dim cellcount As Double
Dim Sheet
cellcount = 0
For Each Sheet In ActiveWorkbook.Worksheets
cellcount = cellcount + Sheet.UsedRange.Cells.Count
Next


Dim message As String
message = "Total cells in workbook: " & vbCrLf & FormatNumber(Round(cellcount), 0) &
vbCrLf
message = message & "Approximate max memory required (at 400 bytes per cell): " &
vbCrLf
message = message & FormatNumber(Round(cellcount * 400), 0) & " bytes " & vbCrLf
message = message & Math.Round(((cellcount / 1024 / 1024 / 1024) * 400), 2) & "
gigabytes"
MsgBox (message)

End Sub
```

Best Practices

Avoid Referenceing Empty Cells

As noted previously, this cell will occupy up to 400 bytes of memory because several other objects have to be instantiated to hold the cell's attributes. Consequently, you should avoid looping through cells that might be empty, since doing so will create Cell objects that you may not need.

Use ExcelApplication before ExcelTemplate

To conserve memory, we recommend pre-processing a partial template with the ExcelApplication before passing the file to ExcelTemplate for importing data. See Preprocessing vs. Postprocessing

Cache Frequently

If you have large reports which are requested often but whose data changes infrequently, you may want to consider using ExcelWriter to generate the document once and then stream it to multiple users. This may be appropriate if the data is changed on a predictable schedule, or it is easy to check to see if the data has been changed. If you want the user to see the most recent data, then you will need to know when new data is available so you can regenerate the report.

If the report contains sensitive data, you will want to take security precautions when using this approach. Since the report will already be generated and saved to disk, it could potentially be easier for an intruder to gain access to it. You may want to use a second server that is not public facing to act as a file server. This file server would only grant access to the report when it is requested by your web application.

When a user requests an Excel document, you can first check to see if there is new data. If there is, you can generate a new report and save it to disk. You can then stream that file to the user. We have a KB article which describes how to stream a file to the user that was previously saved to the disk with ExcelWriter.

Apply Styles to Columns and Rows, not Cells and Areas

Every time Cell.Style is accessed, ExcelWriter instantiates a separate Style object. To reduce file size bloating, we recommend using Global styles and setting styles on groups of cells. For details, see Effective Use of Styles

Applying a style to an Area also creates style objects for each individual cell in the Area. However, if you apply a style to a column or row, there will be only one formatting record for the entire column or row, which is much more efficient. Use ColumnProperties.Style or RowProperties.Style

Use InsertRows or InsertColumns instead of InsertRow or InsertColumn

The Worksheet class has InsertRows, InsertColumns, InsertRow, and InsertColumn methods. If you are only inserting one row, then you should use InsertRow; however, if you are inserting multiple rows, you should make one call to InsertRows and pass the number of rows that you want to insert. The same applies for columns. For example:

```
 //Where you want to insert new rows
int atRow = 10;

//Determine how many rows you'll need to insert
int numRowsToInsert = 4;

//Insert the desired number of rows. This will be much faster
//than multiple calls to InsertRow.
ws.InsertRows(atRow, numRowsToInsert);
```

```
 'Where you want to insert new rows
Dim atRow As Integer = 10

'Determine how many rows you'll need to insert
Dim numRowsToInsert As Integer = 4

'Insert the desired number of rows. This will be much faster
'than multiple calls to InsertRow.
ws.InsertRows(atRow, numRowsToInsert)
```

Avoid calling AutoFitWidth on a lot of data

The ColumnProperties.AutoFitWidth and Area.AutoFitWidth methods are useful for making a column exactly wide enough to fit its contents. However, when there is a lot of data they can take a long time to execute, because they have to go through each row of data and calculate the width of the Cell's contents. Consequently, it is best to use only for columns with small amounts of data that you want to make sure are spaced correctly.

If you have a large area that you want to autofit, and do not know beforehand how wide the cell contents might be, it will be much faster to find the longest string and then set the width of the columns in characters. For example, if you have several thousand rows, then you could say:

```
// POSSIBLE CORRECT CODE:

DataTable dts = GetData(Page.MapPath(@"data\PersonsInfoV2.csv"));

// Column on which the custom autofit starts, to be set by user.
int startingColumn = int.Parse(dts.Rows[0][9].ToString());

// Column on which the custom autofit ends, to be set by user.  (Should be set to the
row AFTER the last column to be autofit.)
int endColumn = int.Parse(dts.Rows[1][9].ToString());

// Array of the longest variables in each column.
int[] longest;

// Temporary variable for keeping track of current location
int temp = 0;

// The loop to check on the width of each column.  Only start if the given starting
and ending places are valid.
if ((startingColumn < dts.Columns.Count)&&(endColumn <
dts.Columns.Count)&&(startingColumn < endColumn))
{
    // Find the size of the array, based on the starting location.
    longest =  new int[endColumn - startingColumn];

    // For each row in the DataTable...
    foreach (DataRow row in dts.Rows)
    {
        // For each column, based on that row...
        for (int i = startingColumn; i < endColumn; i++)
        {
            // Find the temporary length of that row.
            temp = row[i].ToString().Length;

            // Is it the longest in the column so far?  If yes, set it as such.
(Longest subtracts startColumn from index to keep starting index at 0.)
            if (temp > longest[i-startingColumn])
            {
                longest[i-startingColumn] = temp;
            }
        }
    }

    // After looping through, set the width of each column to the longest.
    // You can change this function to change where in the output file
    for (int i = startingColumn; i < endColumn; i++)
    {
        ColumnProperties columnProperties;
        columnProperties = ws.GetColumnProperties(i);
        columnProperties.WidthInChars = longest[i-startingColumn];
    }
}
```

While this will not be as accurate as AutoFitWidth, it will be significantly faster.

Use DataReaders instead of DataTables

The Worksheet.ImportData method takes several different kinds of data types. DataReaders will read data directly from your data source into ExcelApplication's object model, and consequently use less memory and time than other data types. DataTables, DataViews, and arrays must all create an object in memory which contains the data before you can pass that object to ExcelApplication. ExcelApplication will then import the data into its object model. As a result, the data is stored in memory twice for these data types, as opposed to the DataReader where it is stored only once.

DataTables, DataViews, and arrays all use approximately the same amount of memory; however, there is a slight difference in speed. Two dimensional and jagged object arrays will be imported slightly faster, while DataTables and DataViews take approximately 10% longer to import. However, DataReaders remain the fastest, since they do not need to read the data into an object before importing it into ExcelApplication.

Use the newest version OfficeWriter

We are always working to improve the efficiency of ExcelWriter. Make sure to use the latest version of the product to take advantage of these improvements. See the OfficeWriter Change Log for more details.

Example Code

When streamlining a report, it is very important to know which performance issues tend to be the most problematic and the best practices for dealing with those issues.

Below is a collection of common errors that may negatively affect performance, as well as examples of best practices for each situation. There are a number of issues that can slow performance. The examples mention which aspect of performance they are most related to:

- Time based issues, where a report is taking longer to process than would seem normal, yet otherwise runs fine.
- Memory based issues, where the report seems to be using more of the systems resources then necessary.
- Overlap between these two issues is very common; when experiencing performance issues, it is recommended that all of the samples be considered.

Best Practices: Memory Related Performance Issues
Solutions for reports that are taking more memory than it seems they ought to be. Memory Performance

Best Practices: Time Related Performance Issues
Solutions for reports that take longer to process than seems normal. Time Performance