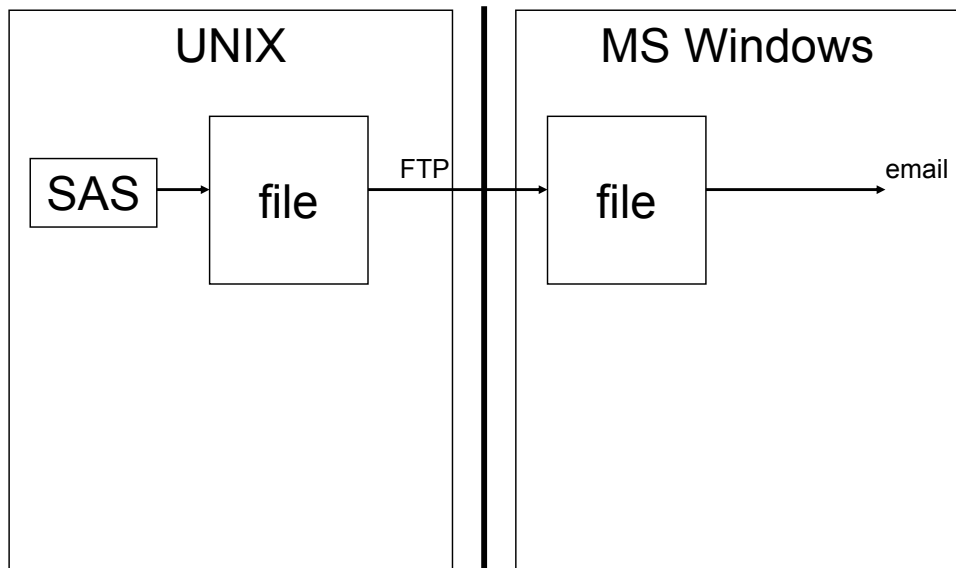


Compressing Output Files in a SAS® job, on a UNIX platform

Leslie J. Somos
Great Works Informatics, LLC

Original situation



2

© 2008 by Great Works Informatics, LLC

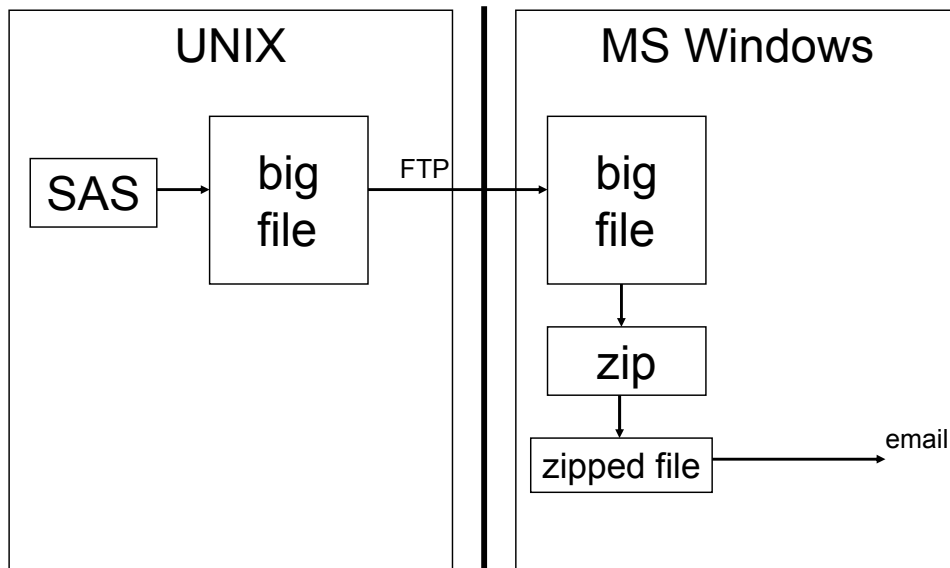
Done for multiple clients.

No problem, until file sometimes is bigger for clients with more data.

As file gets bigger,

FTP from UNIX to MS Windows takes longer,
and email administrators get unhappy.

Original situation



3

© 2008 by Great Works Informatics, LLC

We (manually) zip the file to not annoy the email administrators.

Everything is fine until we hit a wall,
with one particular request for a multi-year extract,
and run out of UNIX space to create the file.

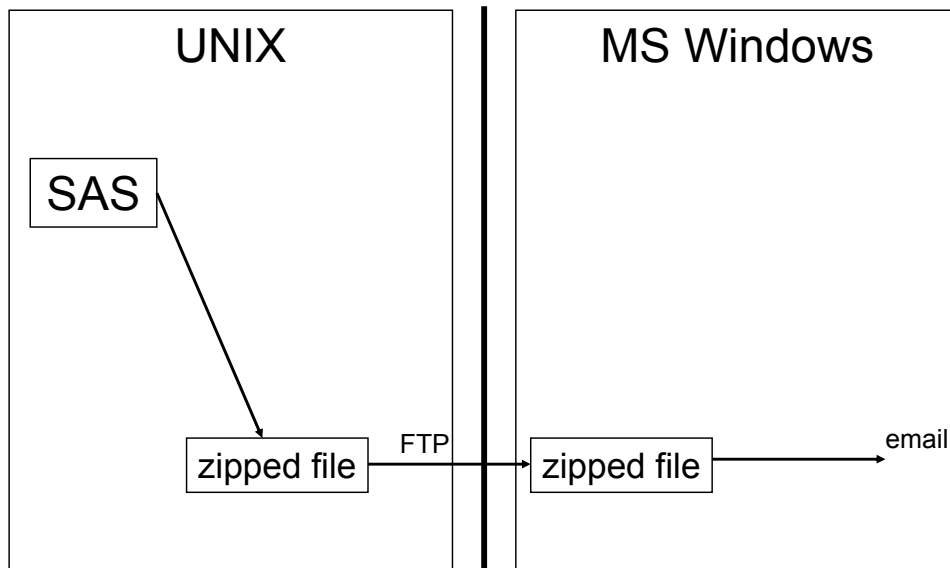
(Two million records, fixed length, record length 2000 bytes,
the resulting file would have been ~4GBytes.

Would have been -- if there had been enough UNIX space.)

--

In each case, after the file is created,
we don't do anything further with it on UNIX,
so it would not be a problem if we could somehow get SAS to write out an already-zipped file.

Goal



4

© 2008 by Great Works Informatics, LLC

So this picture shows what we want to happen -- have SAS write an already-compressed file, which takes up less disk space and also takes less time to FTP from UNIX over to MS Windows.

(4GBytes produces a zipped file ~74Mbytes.
FTP from UNIX to MS Windows ~22 minutes.)

Items we will touch on

- 'pipe' access method of FILENAME statement
- 'nobs=' and 'point=' options of SET statement
- compilation phase v. execution phase of DATA Step

Coding Conventions

- SAS keywords in lowercase
- User-chosen words in uppercase
- Spaces supplied even where not required by the syntax
- Optional period after a macro variable always supplied

Original code

```
filename OUT
  'BIG.TXT'
  lrecl=32760 ;
ordinary file

data _null_ ;
  set OURDATA ;
  file OUT ;
  put <field>...<field> ;
run ;
```

Modify the "filename" statement to use an 'unnamed pipe' to write the data to the "compress" program, which then writes a compressed version to disk.

The full-size file only ever exists on-the-fly, as it flows through the OUT fileref.

Only the compressed version of the file ever exists on disk, so peak disk space usage is less.

Modified code

```
filename OUT pipe  
  'compress > NOTSOBIG.Z'  
  lrecl=32760 ;
```

"unnamed pipe"

*Program compress reads its STDIN, receives data written to filename OUT;
and writes to its STDOUT, which is redirected to NOTSOBIG.Z.*

```
data _null_ ;  
  set OURDATA ;  
  file OUT ;  
  put <field>...<field> ;  
run ;
```

Modify the "filename" statement to use an 'unnamed pipe' to write the data to the "compress" program, which then writes a compressed version to disk.

The full-size file only ever exists on-the-fly, as it flows through the OUT fileref.

Only the compressed version of the file ever exists on disk, so peak disk space usage is less.

Original code :: Modified code

```
filename OUT  
'BIG.TXT'  
lrecl=32760 ;
```

ordinary file

```
data _null_ ;  
  set OURDATA ;  
  file OUT ;  
  put <field>...<field> ;  
run ;
```

```
filename OUT pipe  
'compress > NOTSOBIG.Z'  
lrecl=32760 ;
```

"unnamed pipe"

```
data _null_ ;  
  set OURDATA ;  
  file OUT ;  
  put <field>...<field> ;  
run ;
```

*No changes to **data** step, only to **filename**.
When you uncompress, supply name **BIG.TXT**.*

A simple modification: No changes were necessary to the data step, only to the filename statement.

Since the "compress" command only received a bytestream, it couldn't and didn't record any file name within the compressed file.

When the file is subsequently uncompressed, the program will prompt for a file name to be supplied.

Original code

```
filename OUT
  'BIG.TXT'
  lrecl=32760 ;

data _null_ ;
  set ONEDATA ;
  file OUT ;
  put <field>...<field> ;
run ;

data _null_ ;
  set TWODATA ;
  file OUT mod ;
  put <field>...<field> ;
run ;
```

Slightly more complex code, closer to the actual original code -- multiple data steps write to the same output file.

(Original code had six data steps, two are sufficient to demonstrate the issues we will discuss.)

Each data step after the first one appends to the output file, by using the "mod" option of the file statement.

Original code :: Modified code

```
filename OUT  
'BIG.TXT'  
lrecl=32760 ;
```

```
data _null_ ;  
  set ONEDATA ;  
  file OUT ;  
  put <field>...<field> ;  
run ;
```

```
data _null_ ;  
  set TWODATA ;  
  file OUT mod ;  
  put <field>...<field> ;  
run ;
```

```
filename OUT pipe  
'compress > NOTSOBIG.Z'  
lrecl=32760 ;
```

```
data _null_ ;  
  set ONEDATA ;  
  file OUT ;  
  put <field>...<field> ;  
run ;
```

```
data _null_ ;  
  set TWODATA ;  
  file OUT mod ;  
  put <field>...<field> ;  
run ;
```

*No changes to **data** steps – problem! ⚡
Program **compress** gets called a 2nd time,
and overwrites **NOTSOBIG.Z**.
Only **TWODATA** is in the resulting file.*

If we follow the pattern we know, we modify just the "filename" statement.

Each data step starts the "compress" program anew, and overwrites the previous content of the output file NOTSOBIG.Z. ⚡

The "mod" option of the "file" statement has no effect here, when it refers to an unnamed pipe.

Original code :: Modified code

<pre>filename OUT 'BIG.TXT' lrecl=32760 ; data _null_ ; set ONEDATA ; file OUT ; put <field>...<field> ; run ; data _null_ ; set TWODATA ; file OUT mod ; put <field>...<field> ; run ;</pre>	<pre>filename OUT pipe 'compress > NOTSOBIG.Z' lrecl=32760 ; data _null_ ; file OUT ; set ONEDATA nobs=CONSTNOBS1 ; ⚡ do POINTVAR = 1 to CONSTNOBS1 ; set ONEDATA point=POINTVAR ; put <field>...<field> ; end ; set TWODATA nobs=CONSTNOBS2 ; do POINTVAR = 1 to CONSTNOBS2 ; set TWODATA point=POINTVAR ; put <field>...<field> ; end ; stop ; /*<=REMEMBER!*/ run ; <i>Only one data step, one call to compress.</i> <i>Problem if ONEDATA has zero observations. ⚡</i></pre>
---	--

12

© 2008 by Great Works Informatics, LLC

Solution attempt: Combine the multiple data steps into one single data step, so there is only one invocation of the "compress" program.

[When you take over from the normal SAS data cycle, you also have to include logic to stop execution.]

Note -- the single variable "POINTVAR" is used in multiple loops with no problem.

The two different "nobs=" variables in the two separate "set" statements must be different from each other, else one interferes with the other at compile time.

Works in most cases, but:

If any data set before the last has zero observations, execution stops when its "set" statement is executed, and the output file is incomplete. ⚡

Original code :: Modified code

```
filename OUT  
'BIG.TXT'  
lrecl=32760 ;
```

```
data _null_ ;  
  set ONEDATA ;  
  file OUT ;  
  put <field>...<field> ;  
run ;
```

```
data _null_ ;  
  set TWODATA ;  
  file OUT mod ;  
  put <field>...<field> ;  
run ;
```

```
filename OUT pipe  
'compress > NOTSOBIG.Z'  
lrecl=32760 ;
```

```
data _null_ ;  
  file OUT ;  
  if 0 then  
    set ONEDATA nobs=CONSTNOBS1 ;  
  do POINTVAR = 1 to CONSTNOBS1 ;  
    set ONEDATA point=POINTVAR ;  
    put <field>...<field> ;  
  end ;  
  if 0 then  
    set TWODATA nobs=CONSTNOBS2 ;  
  do POINTVAR = 1 to CONSTNOBS2 ;  
    set TWODATA point=POINTVAR ;  
    put <field>...<field> ;  
  end ;
```

```
stop ; /*<=REMEMBER!*/  
run ;
```

The "set" statements where the values of CONSTNOBS1 and CONSTNOBS2 are set have their effect at compile time, not at execution time.

So, they don't ever have to be executed, they simply have to be present at compile time.

They could be anywhere within the data step, they simply have to be present at compile time.

As an alternative to prefixing each "set ... nobs=" statement with "if 0" or "if 1 = 2" etc.,

we could move them to after the "stop" statement.

Original code :: Modified code

```
filename OUT  
'BIG.TXT'  
lrecl=32760 ;
```

```
data _null_ ;  
  set ONEDATA ;  
  file OUT ;  
  put <field>...<field> ;  
run ;
```

```
data _null_ ;  
  set TWODATA ;  
  file OUT mod ;  
  put <field>...<field> ;  
run ;
```

```
filename OUT pipe  
'compress > NOTSOBIG.Z'  
lrecl=32760 ;
```

```
data _null_ ;  
  file OUT ;
```

```
do POINTVAR = 1 to CONSTNOBS1 ;  
  set ONEDATA point=POINTVAR  
                      nobs=CONSTNOBS1 ;  
  put <field>...<field> ;  
end ;
```

```
do POINTVAR = 1 to CONSTNOBS2 ;  
  set TWODATA point=POINTVAR  
                      nobs=CONSTNOBS2 ;  
  put <field>...<field> ;  
end ;
```

```
stop ; /*<=REMEMBER!*/  
run ;
```

The compile-time "nobs=" and the execution-time "point=" options of the "set" statement can both be present on a single "set" statement.

And, if any data set before the last has zero observations,

its "set" statement is not actually executed at run time because its "nobs=" variable is set to zero at compile time

and that "do" loop reduces to "do POINTVAR = 1 to 0 ;"

and execution does not enter that "do" loop.

DATA Step -- compilation phase v. execution phase

<http://v8doc.sas.com>

SAS OnlineDoc

. Base SAS Software

. . SAS Language Reference: Concepts

. . . DATA Step Concepts

. . . . DATA Step Processing

. Overview of DATA Step Processing

Flow of Action

When you submit a DATA step for execution, it is first compiled and then executed.

.

SET

<http://v8doc.sas.com>

SAS OnlineDoc

. Base SAS Software

. . SAS Language Reference: Dictionary

. . . Dictionary of Language Elements

. . . . Statements

. SET

NOBS=variable

At compilation time, SAS reads the descriptor portion of each data set and assigns the value of the NOBS= variable automatically. Thus, you can refer to the NOBS= variable before the SET statement.

POINT=variable

POINT= causes the SET statement to use random (direct) access to read a SAS data set.

CAUTION:

Continuous loops can occur when you use the POINT= option.

When you use the POINT= option, you must include a STOP statement to stop DATA step processing, programming logic that checks for an invalid value of the POINT= variable, or both.

Reading from and Writing to UNIX Commands (PIPE)

<http://v8doc.sas.com>

SAS OnlineDoc

. Base SAS Software

. . Host Specific Information

. . . UNIX Environments (Companion)

. . . . Running the SAS System Under UNIX

. Using External Files and Devices

. Reading from and Writing to UNIX Commands (PIPE)

FILENAME *fileref* PIPE '*UNIX-command*' <*options*>;

Under UNIX, you can use the FILENAME statement to assign filerefs not only to external files and I/O devices, but also to a pipe. Pipes enable your SAS application to receive input from any UNIX command that writes to standard output and to route output to any UNIX command that reads from standard input.

Using Unnamed Pipes (MS Windows)

<http://v8doc.sas.com>

SAS OnlineDoc

. Base SAS Software

.. Host Specific Information

... Microsoft Windows Environment (Companion)

.... Using SAS with Other Windows Applications

..... Using Unnamed and Named Pipes

..... Using Unnamed Pipes

FILENAME *fileref* PIPE '*program-name*' *option-list*

Example:

```
filename DIR pipe 'dir /?' ;
data _null_ ;
  infile DIR ;
  input ;
  put _infile_ ;
run ;
```

```
NOTE: The infile DIR is:
       Unnamed Pipe Access Device,
       PROCESS=dir /?,RECFM=V,LRECL=256
Displays a list of files and subdirectories in
a directory.
...
NOTE: 38 records were read from the infile DIR.
       The minimum record length was 0.
       The maximum record length was 75.
NOTE: DATA statement used (Total process time):
       real time           0.09 seconds
       cpu time            0.03 seconds
```

Unfortunately, I couldn't find any compression program under MS Windows which would read a file to be compressed from its standard input.

Questions?

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 registered trademarks or trademarks of their respective companies.