SAS® GLOBAL FORUM 2019

USERS PROGRAM

APRIL 28 – MAY 1, 2019 | DALLAS, TX

Application of Command-Line Interface Tools to Workflows Based on SAS®

Boone Tensuda and Shahriar Khosravi

BMO Financial Group, Canada

Abstract

This e-poster showcases some of the potential applications of the command-line interface tools typically found within operating systems based on Linux to workflows built on SAS®. By executing SAS from the command line, a plethora of tools become available to the user. Some tools that are explored in this presentation are Bash, rsync, and Make. We demonstrate the application of these tools through individual examples for tasks such as automated code execution, scheduling, backup, and parallelization with SAS. The integration of the tools for advanced applications, such as automated software testing and dynamic parallelization, are also presented. Practical examples highlight the simplicity of implementation and potential efficiency improvements associated with these tools for typical SAS applications.

Motivation

Executing SAS scripts/workflows from the command line interface (CLI) allows the user to leverage 100s of supporting CLI tools. These tools accomplish helpful tasks such as: running processes in the background, scheduling processes, creating “sandbox” environments, automation, parallelization, and text parsing. Although these features overlap with various SAS products to some extent, they have several advantages. They are free and accessible, easy to integrate, stable, and purpose driven, which leads to simple use and integration with SAS driven workflows.

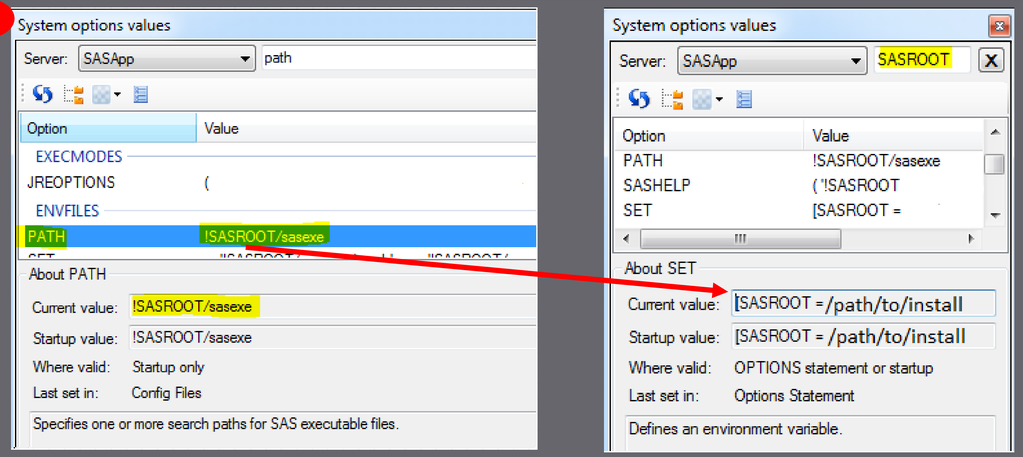
How to Run SAS from the CLI

1. Gain access to the system on which SAS is installed (e.g. using **ssh**).
2. Determine the location of the SAS executable file. You may use the SAS “PATH” option to determine this (see example to the right using SAS Enterprise Guide).
3. Run the SAS command found in step 2 followed by the SAS script name.
4. Two files will be output; .log and .lst, which contain the resulting SAS log and report, respectively.

1

```
>>ssh username@domain
```

2



3

```
>> /path/to/install/sas demo.sas
```

4

```
>>ls
demo.log  demo.lst  demo.sas
```

Bash Scripting

- Combine SAS scripts and Linux commands into one “master” script.
- Bash scripts are clear, repeatable, transferable, and version controllable (e.g. with Git).
- The pipe character “|” in the last line of the script is used to send the output of one command directly to another command.

Example 1

```
#!/bin/bash

echo "This is a BASH script"
command1
/path/to/install/sas demo.sas
command2
generate_output | consume_output
```

Application of Command-Line Interface Tools to Workflows Based on SAS®

Boone Tensuda and Shahriar Khosravi

BMO Financial Group, Canada

Remote Execution via Background Processes

- Big data operations using SAS often require the user to submit a query or script to the server via SAS Enterprise Guide™ for it to run overnight.
- This poses a challenge for the user because if the SAS Enterprise Guide session is terminated for any reason (e.g. local power outage or a surprise restart), then the query will be killed, and the user may lose a day of work.
- One solution to mitigate this risk is to use the "**nohup**" tool in Linux in order to execute a SAS query .This will submit the process to the background, such that it will be executed on the server regardless of what happens to the user's local machine:

Example 1

```
>> nohup /path/to/install/sas demo.sas &
```

Scheduling Jobs

- Users can easily schedule jobs to be executed at specific times.
- For example, if the user would like to schedule the "demo.sas" query at 1:30pm today, all that is required is:

Example 2

```
>> at 1:30pm today
at> /path/to/install/sas demo.sas
at> <EOT>
job 1 at 2019-04-30 13:30
```

- In the above example, "<EOT>" marks the end of the list of jobs to be executed, and could be inserted by Ctrl+D.
- Any number of commands may be entered between the "**at**" command and the "<EOT>".

Sandboxing – Part 1

- When running queries on large datasets, it may be helpful to limit the amount of memory (RAM and ROM) and CPU resources that a SAS script has access to.
- An obvious and important reason for limiting resources is to avoid causing widespread server outages due to possible bugs in the script.
- One useful tool is "**prlimit**" that allows the user to limit output file size, for instance:

Example 3

```
>> prlimit --fsize=100000000 /path/to/install/sas test.sas
```

Parsing SAS Log Files with AWK

- A powerful CLI pattern scanning and processing language for extracting information and data from raw text files is "**awk**".
- One useful application of "**awk**" in SAS programming is extracting notes, warnings, or errors from large or numerous log files produced by complex SAS scripts.
- Log files from SAS (or other applications) are typically structured in a predictable way, as demonstrated in Example 4.

Example 4

```
NOTE :      This is a note
NOTE :      This is another note
WARNING :   This is a warning
WARNING :   Pay attention to warnings
ERROR :     SEG FAULT
NOTE :      I'm sorry
```

$1 **$2**

- In this example, the columns of text on the left- and right-hand sides may be referenced in "**awk**" by Column Identifiers "**$1**" and "**$2**", respectively. Subsequent columns may similarly be referenced by "**$3**", "**$4**", etc.
- Each column of text is separated from the next by a Field Separator. In Example 4, the Field Separator is the colon punctuation mark (:).
- The anatomy of a one-line "**awk**" command is shown below in Example 5.

Example 5

```
>> awk 'BEGIN(FS=":")(condition){print $x}' filename.log
```

Field **S**eparator — Column Identifier

- Example 6 is a one-line "**awk**" command to grab only warnings from the log.
- In this example, the condition is if the first column of the text is equal to WARNING. As such, the command will print the second column only if the first column is equal to WARNING. The output of the command is also shown.

Example 6

```
awk 'BEGIN(FS=":")($1 == "WARNING"){print $2}' filename.log
```

```
NOTE :      This is a note
NOTE :      This is another note
WARNING :   This is a warning
WARNING :   Pay attention to warnings
ERROR :     SEG FAULT
NOTE :      I'm sorry
```

$1 **$2**

→

```
This is a warning
Pay attention to warnings
```

Application of Command-Line Interface Tools to Workflows Based on SAS®

Boone Tensuda and Shahriar Khosravi

BMO Financial Group, Canada

Automated Testing

- Regression/consistency testing is often an important requirement in software release management cycles.
- For instance, comparing the output of two different code bases contained in different Git tags or branches may be required to ensure that both code bases produce consistent results.
- Without any automation, this may be a cumbersome task since it involves manually switching between Git tags or branches before executing each code base.
- This may be automated by coding the Git commands inside a Linux Bash script, as demonstrated in the Example 1.
- In this example, “GITPATH” and “SCRIPTPATH” are the paths to the location of the Git repository and the test script (i.e. the SAS script containing the code for regression/consistency tests), respectively.
- Standard Git commands may be used inside of the Bash script to checkout different branches or a specific tag.
- The next step is to execute the SAS script for the regression/consistency test inside the Bash script, as shown in Example 2.
- The output dataset from the baseline code may be renamed, such that it does not get overwritten by the next execution, as shown in Example 3.
- The next step is to checkout the comparison branch from the Git repository and execute the test script, as shown in Example 4.
- The new output may be renamed to distinguish it from the baseline results.
- Finally, another SAS script may be called to compare the two outputs and publish the results in a report.

Example 1

```
#define the path to the Git repository
GITPATH="/path/to/Git/repo"
#define the path to the test script
SCRIPTPATH="/path/to/test/script"

#checkout the baseline branch of the code
cd $GITPATH
git checkout baseline_branch
cd $SCRIPTPATH
```

Example 2

```
#run the regression test using baseline code base
/path/to/install/sas regression_test.sas
```

Example 3

```
#rename the output from the test script
mv regression_output.sas7bdat baseline_output.sas7bdat
```

Example 4

```
#checkout the comparison branch of the code
cd $GITPATH
git checkout comparison_branch
cd $SCRIPTPATH

#run the regression test using comparison code base
/path/to/install/sas regression_test.sas

#rename the output from the test script
mv regression_output.sas7bdat compare_output.sas7bdat

#run a different SAS script that compares the outputs
/path/to/install/sas compare_results.sas
```

Sandboxing – Part 2

- Another sandboxing strategy would be to use a combination of CLI tools inside a Bash script in order to monitor the size of the current SAS work directory.
- A useful tool for monitoring the size of the work directory inside a Bash script is “**du**”.
- In the example below, “**du**” is used along with “**tail**” and “**awk**” in order to extract only the folder size.

Example 4

```
>> du -cs /path/to/saswork/ | tail -n1 | awk '{print $1}';
```

- The current SAS work directory and its size can be obtained using a combination of command line tools inside a Bash script, as demonstrated below:

Example 5

```
WORK_DIR=`ls -l /proc/$!/fd | grep saswork |\
awk 'BEGIN{FS="/"}; {print "/"$2"/"$3"/"$4"/"$5}' | head -n 1`
WORK_DIR_SIZE=`du -cs $WORK_DIR | tail -n1 | awk '{print $1}'`;
```

- In the same Bash script, triggers may be defined for violating a user-specified workspace size and time limit.
- The triggers could then be used in a loop in order to check the directory size and time limit periodically, as demonstrated below:

Example 6

```
while [[ $PID_END_TRIGGER -eq 0 && $TIME_LIMIT_TRIGGER -eq 0 && $WORK_LIMIT_TRIGGER -eq 0 ]]
do

        WORK_DIR_SIZE=`du -cs $WORK_DIR | tail -n1 | awk '{print $1}'`;
        echo "current work dir size: $WORK_DIR_SIZE"

        if [ $(date +"%s") -ge $(($START_TIME + $INTERVAL)) ]
        then
                TIME_LIMIT_TRIGGER=1;
        else
                TIME_LIMIT_TRIGGER=0;
        fi

        if [ $WORK_DIR_SIZE -ge $WORK_MAX ]
        then
                WORK_LIMIT_TRIGGER=1;
        else
                WORK_LIMIT_TRIGGER=0;
        fi

        if [ $(ps -p $! | wc -l) -eq 1 ]
        then
                PID_END_TRIGGER=1;
        else
                PID_END_TRIGGER=0;
        fi

        sleep 30;
done
```

- In this case, if either of the triggers are set equal to 1 inside the “while” loop, the SAS process could be killed within the same Bash script:

Example 7

```
echo "";
echo "process ended trigger = $PID_END_TRIGGER";
echo "time limit trigger = $TIME_LIMIT_TRIGGER";
echo "size limit trigger = $WORK_LIMIT_TRIGGER";

echo "killing $!";
kill -15 $!;
```

Application of Command-Line Interface Tools to Workflows Based on SAS®

Boone Tensuda and Shahriar Khosravi

BMO Financial Group, Canada

Make

- Make is a CLI tool that will automatically optimize, organize, and parallelize your SAS workflows.
- Simply write one extra file (Makefile), see below.
- The format of the command is "**make**" followed by the desired target. The "**-j**" option can be used to specify the number of CPUs.

Example 1

```
>>make all -j12
```

Makefile

- Has a very simple structure; write the input, output (target), and how to go from input to output (function). In the example below "all" is a "dummy" target, which means a file called "all" is not actually created.

Example 2

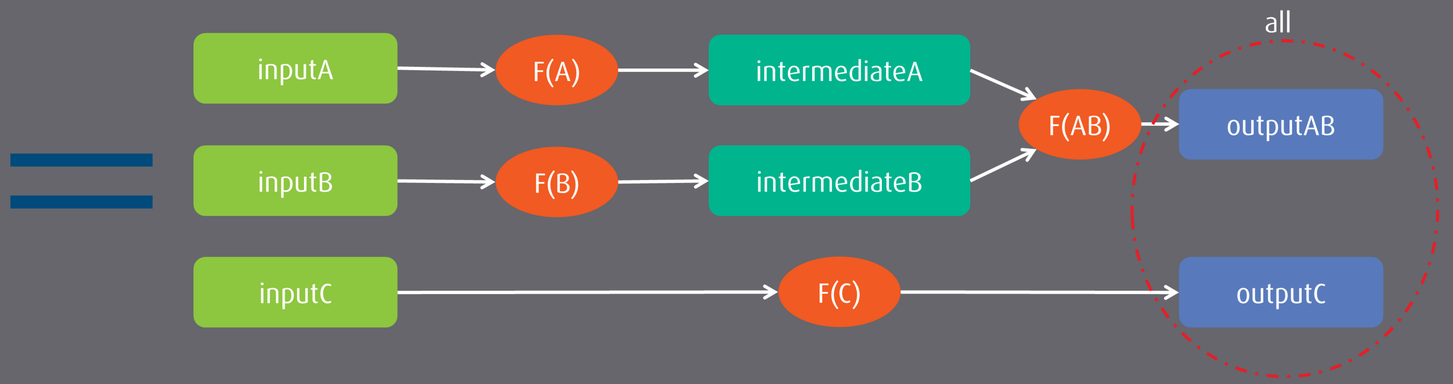
```
all: outputAB outputC

outputAB: intermediateA intermediateB
        functionAB

outputC: inputC
        functionC

intermedateA: inputA
        functionA

intermidateB: inputB
        functionB
```



Make Features

- Make can distinguish between different sections of workflows. For example, "all: outputAB outputC" in the example below will only execute functions required to generate those outputs (in this case all functions).
- Make will **not perform redundant work**, for example, if only inputC has changed, only F(C) will be executed.
- Make can **combine inputs/outputs of different programs into one workflow,** for example F(A) and F(B) could be SAS scripts, while F(AB) is a Python script.
- Make will **automatically parallelize** the workflow (with the **–j** option). In the example below F(A), F(B), and F(C) will be executed independently and distributed to different processors (F(AB) will not be since it depends on the output of F(A) and F(B)).

Application of Command-Line Interface Tools to Workflows Based on SAS®

Boone Tensuda and Shahriar Khosravi

BMO Financial Group, Canada

Make Application

- This example workflow contains several stages; extracting data from some source location, performing analysis, and testing the output.
- The extraction and analysis of source_A and source_B are independent, however the final results are expected to be equivalent.

Extracting Data with rsync

- **rsync** is a CLI tool for efficient copying and archiving of files.
- It has the power to prevent unnecessary data transfer, using file hashes, modification times, and file sizes.

Example 1

```
SAS=/path/to/install/sas

extract: ./land/land_A.csv ./land/land_B.csv
analysis: output_a.sas7bdat output_b.sas7bdat
all: extract analysis

test: run_test

./land/land_A.csv : ./source/source_A.csv
        rsync ./source/source_A.csv ./land/land_A.csv

./land/land_B.csv : ./source/source_B.csv
        rsync ./source/source_B.csv ./land/land_B.csv

output_a.sas7bdat : ./land/land_A.csv
        -$(SAS) demo_A.sas

output_b.sas7bdat : ./land/land_B.csv
        -$(SAS) demo_B.sas

run_test:
        -$(SAS) -nocenter test.sas
        grep "NOTE:" test.lst
```

```
PROC COMPARE
        base=output_A
        compare=output_B;
RUN;
```

Extract Optimization

- **Make** will automatically check the modification time of the “input” files to prevent unnecessary work.
- Example: The “extract” dummy target is executed, but none of the source files have changed:

Example 2

```
>>make extract
make: Nothing to be done for `extract'.
```

Parallelization of Analysis

- Even in this simple example, the “**-j**” option to parallelize leads to twice as fast execution of the extraction and analysis workflows:

Example 3

Serial execution

```
>>time make all
rsync ./source/source_A.csv ./land/land_A.csv
rsync ./source/source_B.csv ./land/land_B.csv
/path/to/install/sas    demo_A.sas
/path/to/install/sas    demo_B.sas

real    1m10.372s
user    1m3.672s
sys     0m20.240s
```

Parallel execution

```
>>time make all -j2
rsync ./source/source_A.csv ./land/land_A.csv
rsync ./source/source_B.csv ./land/land_B.csv
/path/to/install/sas    demo_A.sas
/path/to/install/sas    demo_B.sas

real    0m35.183s
user    1m4.345s
sys     0m19.396s
```

Automated Testing

- The “test” workflow will run a SAS script which compares the output_a and output_b SAS datasets and displays the “NOTE:” lines from the PROC COMPARE report.

Example 4

```
>>make test
/path/to/install/sas   -nocenter test.sas
grep "NOTE:" test.lst
NOTE: No unequal values were found. All value
s compared are exactly equal.  Passed ✓
```