# The inner workings of the datastep

By Mathieu Gaouette

Videotron

# Plan

- Introduction
- The base
- The base behind the scene
- Control in the datastep
- A side by side compare with Proc SQL

# Introduction

- Most of you probably have been introduced to SAS through Proc SQL.
- Unless you have been taught (or have read) about the datastep, most of you probably very rarely use it.
- Do you think you could name one thing that can be done with a datastep that can't be performed with a proc SQL?
- Knowledge of how the datastep is being processed by SAS is key in using it wisely.

# The base

The anatomy of the datastep is fairly simple:

Data **<table(s) to create >** ;

**<Stuff! ( input definition, functions, calculations, ...)>**

Run ;

# The base - Input

| | SAS table input | Flat file input |
|---|---|---|
| Identification | **Set or merge statement**<br><set\|merge> *table1 table2* …<br>[options]; | **Infile statement**<br><infile> *external-file* [options]; |
| Input instruction | Same statement | Input statement |

- Reading in data makes the datastep loop over as long as there is data to read…
  in most cases

# The base - Output

| | SAS table output | Flat file output |
|---|---|---|
| Source definition | **Data statement**<br>*Data table1 table2 … ;* | **File statement**<br>*<File> external-file [options];* |
| Output instruction | *output [table-name] ;* | put statement |

For SAS tables, if no explicit output is used, an implicit ouput statement is executed when the datastep execution hits the « run » statement.

# Behind the scene

- Data step processing order
- Program data vector (PDV)
- Automatic PDV variables
- Detailed step by step example

# Processing the datastep

1. The datastep initiates
2. If required, an input buffer is created
3. A program data vector is created (PDV)
4. The output dataset(s) are created empty

**Only then** is the first line

of the datastep is actually processed.

# Why is that important?

The actual locations of a few key statements are irrelevant in a datastep.

Consider the following datastep:

```
data test_no1 ;
   val_a = 1 ; val_b = 2 ;
   if val_a = 3 then do ;
      drop val_a ;
   end ;
   else if val_b = 3 then do ;
      drop val_b ;
   end ;
run ;
```

None of these two sub sections get executed

```
NOTE: The data set WORK.TEST_NO1 has 1 observations and 0 variables.
NOTE: DATA statement used (Total process time):
      real time                  0.01 seconds
      cpu time                   0.01 seconds
```

# Another example

```
data src_table_1 ;
   val1_a = 1 ; val1_b = 1 ; val1_c = 1 ;
run ;
data src_table_2 ;
   val2_a = 2 ; val2_b = 2 ; val2_c = 2 ;
run ;
data test_no2 ;
   if "&SYSUSERID." eq 'gaouettm' then set src_table_1 ;
   else set src_table_2 ;
run ;
```

```
NOTE: There were 1 observations read from the data set WORK.SRC_TABLE_1.
NOTE: The data set WORK.TEST_NO2 has 1 observations and 6 variables.
NOTE: DATA statement used (Total process time):
      real time          0.01 seconds
      cpu time           0.03 seconds
```

# A closer look at the PDV

- The PDV should be viewed as a draft of your data.
- It contains all of your dataset variables (even dropped variables) plus two system variables :
  1. _N_
  2. _ERROR_
- Knowing about these two system variables can be an asset.

# QUIZ

- What is the minimal possible value of the datastep system variable _N_?

  A) 0

  B) 1

  C) *Is that underscore part of the variable name?*

# _N_

- Contrary to popular belief, this system variable doesn't track the row number being processed.
- "Each time the DATA step loops past the DATA statement, the variable _N_ increments by 1. The value of _N_ represents the number of times the DATA step has iterated." (SAS.com)
- It's actually more: "The value of _N_ represents the number of times the DATA step has iterated **plus one**."

# Typical use of _N_

- Limit the number of iteration in a datastep :

  if _n_ > 1000 then stop ;

- Perform one time task from within the datastep :

```
if _n_ = 1 then do ;
      <code to be executed one time>
end ;
```

- Create an incremental id variable :

  id_key = _n_ ;

# _ERROR_

- is 0 by default but is set to 1 whenever an error is encountered, such as an input data error, a conversion error, or a math error, as in division by 0 or a floating point overflow. You can use the value of this variable to help locate errors in data records and to print an error message to the SAS log. (SAS.com)

# QUIZ

- When a « _ERROR_ » is produced in a datastep, does SAS generates a « WARNING: » and/or « ERROR: » in the log?

A) Yes

B) No

C) *It's complicated.  I'd rather not talk about it*

# What triggers _ERROR_

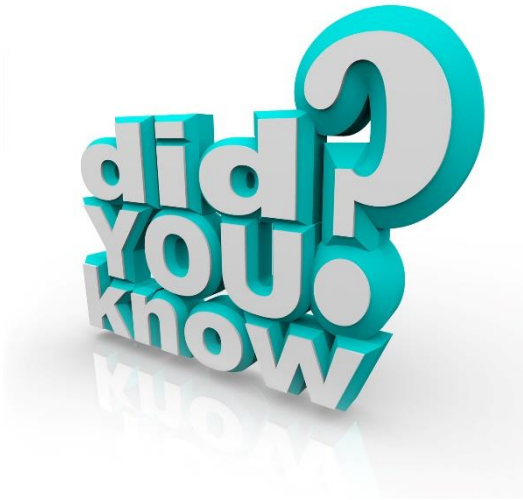
A few comon situations are:

- Divisions by zero
  - only triggers a NOTE in the log
- Invalid array position reference
  - triggers an "ERROR:" in the log
- Invalid value for input/put function
  - only triggers a NOTE in the log

# A note about NOTEs

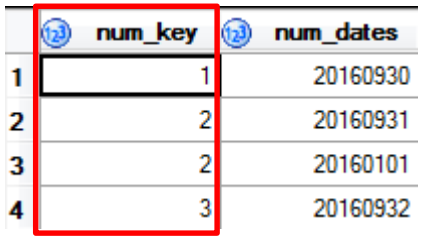
- You can use input with an option that suppresses the errors.
  - A single '?' with a space before the format tells SAS to not print the NOTE.
  - A double '?' with a space before the format will also reset the _ERROR_ value to 0

Ex: n_date = input(c_date,?? yymmdd10.) ;



# Detailed exemple

- Lets start with two simple tables …that share a comon key



| | num_key | num_dates |
|---|---|---|
| 1 | 1 | 20160930 |
| 2 | 2 | 20160931 |
| 3 | 2 | 20160101 |
| 4 | 3 | 20160932 |

| | num_key | char_val |
|---|---|---|
| 1 | 1 | Hello world |
| 2 | 2 | Hello world |
| 3 | 3 | Hello world |
| 4 | 4 | Hello world |
| 5 | 5 | Hello world |

- We wish to merge then and try to convert the « num_dates » into a SAS date.

```
data toto ;
   retain count_obs 0 ;
   merge src_a(in=a) src_b(in=b) ;
   count_obs = count_obs + 1 ;
   by num_key ;
   if a ;
   char_nonsense_date = input(put(num_dates,8.),yymmdd8.) ;
   output ;
run ;
```

# Lets keep an eye on the PDV

| Iteration no 1 | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| count_ obs | a | b | num_ key | num_dates | char_val | char_ nonsense_ date | _ERROR_ | _N_ | Program step (PVD values taken before step) |
| 0 | 0 | 0 | . | . | | . | 0 | 1 | retain count_obs 0 ; |
| 0 | 0 | 0 | . | . | | . | 0 | 1 | merge src_a(in=a) src_b(in=b) ; |
| 0 | 1 | 1 | 1 | 20160930 | Hello world | . | 0 | 1 | count_obs = count_obs + 1 ; |
| 1 | 1 | 1 | 1 | 20160930 | Hello world | . | 0 | 1 | by num_key ; |
| 1 | 1 | 1 | 1 | 20160930 | Hello world | . | 0 | 1 | if a ; |
| 1 | 1 | 1 | 1 | 20160930 | Hello world | . | 0 | 1 | char_nonsense_date = input(put(num_dates,8.),yymmdd8.) ; |
| 1 | 1 | 1 | 1 | 20160930 | Hello world | 20727 | 0 | 1 | output ; |

- Retained count_obs is initialized before the statement is executed.
- Input variables are set to missing until data is read.
- Char_nonsense_date actually gets a decent date value assigned.

# Lets keep an eye on the PDV

| Iteration no 2 | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| count_ obs | a | b | num_ key | num_dates | char_val | char_ nonsense_ date | _ERROR_ | _N_ | Program step (PVD values taken before step) |
| 1 | 1 | 1 | 1 | 20160930 | Hello world | . | 0 | 2 | retain count_obs 0 ; |
| 1 | 1 | 1 | 1 | 20160930 | Hello world | . | 0 | 2 | merge src_a(in=a) src_b(in=b) ; |
| 1 | 1 | 1 | 2 | 20160931 | Hello world | . | 0 | 2 | count_obs = count_obs + 1 ; |
| 2 | 1 | 1 | 2 | 20160931 | Hello world | . | 0 | 2 | by num_key ; |
| 2 | 1 | 1 | 2 | 20160931 | Hello world | . | 0 | 2 | if a ; |
| 2 | 1 | 1 | 2 | 20160931 | Hello world | . | 0 | 2 | char_nonsense_date = input(put(num_dates,8.),yymmdd8.) ; |
| 2 | 1 | 1 | 2 | 20160931 | Hello world | . | 1 | 2 | output ; |

- First row of values are kept in PDV until merge statement is executed.
- Date conversion fails so _ERROR_ is set to 1 and the following note gets displayed in log:

**NOTE: Invalid argument to function INPUT at line 53 column 26.**

# Lets keep an eye on the PDV

| Iteration no 3 | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| count_ obs | a | b | num_ key | num_dates | char_val | char_ nonsense_ date | _ERROR_ | _N_ | Program step (PVD values taken before step) |
| 2 | 1 | 1 | 2 | 20160931 | Hello world | . | 0 | 3 | retain count_obs 0 ; |
| 2 | 1 | 1 | 2 | 20160931 | Hello world | . | 0 | 3 | merge src_a(in=a) src_b(in=b) ; |
| 2 | 1 | 1 | 2 | 20160101 | Hello world | . | 0 | 3 | count_obs = count_obs + 1 ; |
| 3 | 1 | 1 | 2 | 20160101 | Hello world | . | 0 | 3 | by num_key ; |
| 3 | 1 | 1 | 2 | 20160101 | Hello world | . | 0 | 3 | if a ; |
| 3 | 1 | 1 | 2 | 20160101 | Hello world | . | 0 | 3 | char_nonsense_date = input(put(num_dates,8.),yymmdd8.) ; |
| 3 | 1 | 1 | 2 | 20160101 | Hello world | 20454 | 0 | 3 | output ; |

- 3rd iteration starts off fresh with _ERROR_ back to 0.
- Second line of data for num_key 2 read (only the num_dates field changes). The pointer to the table src_b still points to the same row (num_key of 2).

# Lets keep an eye on the PDV

| Iteration no 4 | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| count_ obs | a | b | num_ key | num_dates | char_val | char_ nonsense_ date | _ERROR_ | _N_ | Program step (PVD values taken before step) |
| 3 | 1 | 1 | 2 | 20160101 | Hello world | . | 0 | 4 | retain count_obs 0 ; |
| 3 | 1 | 1 | 2 | 20160101 | Hello world | . | 0 | 4 | merge src_a(in=a) src_b(in=b) ; |
| 3 | 1 | 1 | 3 | 20160932 | Hello world | . | 0 | 4 | count_obs = count_obs + 1 ; |
| 4 | 1 | 1 | 3 | 20160932 | Hello world | . | 0 | 4 | by num_key ; |
| 4 | 1 | 1 | 3 | 20160932 | Hello world | . | 0 | 4 | if a ; |
| 4 | 1 | 1 | 3 | 20160932 | Hello world | . | 0 | 4 | char_nonsense_date = input(put(num_dates,8.),yymmdd8.) ; |
| 4 | 1 | 1 | 3 | 20160932 | Hello world | . | 1 | 4 | output ; |

- This iteration behaves a lot like the second one.
- A new line of data corresponding to a new num_key value is read from both tables
- An error is encountered while converting the bogus date.

# Lets keep an eye on the PDV

| Iteration no 5 | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| count_ obs | a | b | num_ key | num_dates | char_val | char_ nonsense_ date | _ERROR_ | _N_ | Program step (PVD values taken before step) |
| 4 | 1 | 1 | 3 | 20160932 | Hello world | . | 0 | 5 | retain count_obs 0 ; |
| 4 | 1 | 1 | 3 | 20160932 | Hello world | . | 0 | 5 | merge src_a(in=a) src_b(in=b) ; |
| 4 | 0 | 1 | 4 | . | Hello world | . | 0 | 5 | count_obs = count_obs + 1 ; |
| 5 | 0 | 1 | 4 | . | Hello world | . | 0 | 5 | by num_key ; |
| 5 | 0 | 1 | 4 | . | Hello world | . | 0 | 5 | if a ; |
| | | | | | | | | | char_nonsense_date = |
| | | | | | | | | | input(put(num_dates,8.),yymmdd8.) ; |
| | | | | | | | | | output ; |

- _ERROR_ initialized again
- Missing values for variables from table src_a as it does not contain the num_key 4.
- As “in variable” a is equal to 0, iteration stops there.

# Lets keep an eye on the PDV

| Iteration no 6 | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| count_ obs | a | b | num_ key | num_dates | char_val | char_ nonsense_ date | _ERROR_ | _N_ | Program step (PVD values taken before step) |
| 5 | 0 | 1 | 4 | . | Hello world | . | 0 | 6 | retain count_obs 0 ; |
| 5 | 0 | 1 | 4 | . | Hello world | . | 0 | 6 | merge src_a(in=a) src_b(in=b) ; |
| 5 | 0 | 1 | 5 | . | Hello world | . | 0 | 6 | count_obs = count_obs + 1 ; |
| 6 | 0 | 1 | 5 | . | Hello world | . | 0 | 6 | by num_key ; |
| 6 | 0 | 1 | 5 | . | Hello world | . | 0 | 6 | if a ; |
| | | | | | | | | | char_nonsense_date = |
| | | | | | | | | | input(put(num_dates,8.),yymmdd8.) ; |
| | | | | | | | | | output ; |

- Again, missing values for variables from table src_a as it does not contain the num_key 5.
- As “in variable” a is equal to 0, iteration stops there and we are done with the datastep.
- Except we are not!

# Lets keep an eye on the PDV

| Iteration no 7 | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| count_ obs | a | b | num_ key | num_dates | char_val | char_ nonsense_ date | _ERROR_ | _N_ | Program step (PVD values taken before step) |
| 6 | 0 | 1 | 5 | . | Hello world | . | 0 | 7 | retain count_obs 0 ; |
| 6 | 0 | 1 | 5 | . | Hello world | . | 0 | 7 | merge src_a(in=a) src_b(in=b) ; |
| | | | | | | | | | count_obs = count_obs + 1 ; |
| | | | | | | | | | by num_key ; |
| | | | | | | | | | if a ; |
| | | | | | | | | | char_nonsense_date = input(put(num_dates,8.),yymmdd8.) ; |
| | | | | | | | | | output ; |

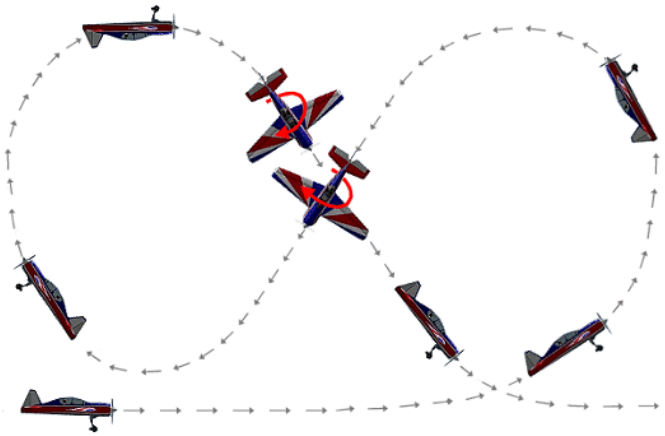
- SAS loops again until it tries to read a new row of data from input files.
- Since SAS can not read any data in, it stops processing the current iteration.

# Control in the datastep

- Conditional processing and loops are huge strengths of the datastep.
- The basic datastep goes from top to bottom one line at a time.
- With loops and conditions, you can execute some statements more

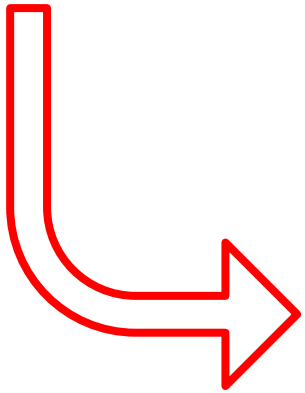
  than once or not at all

  in specific iterations.



# Control in the datastep

| Instruction | Statement |
|---|---|
| Stop processing the current iteration | delete ; |
| Stop processing the current datastep | stop ; |
| Conditional processing | If - then - do |
| Looping | Do, do while, do until |
| Go to specific portion of the datastep | go to statement |



# The datastep


# Proc SQL

# Side by side

| | Data step | Proc SQL |
|---|---|---|
| Joins | requires sorted/indexed input | No requirement* ✓ |
| Unions | YES ✓ | No interleave possible |
| Output | Multiple outputs ✓ | Single output |
| Conditionnal processing | Strong with minimal code ✓ | Strong but with a toll on the complexity |
| Aggregations | Manual and requires sorted input | No real limits ✓ |
| Work usage | Minimal* ✓ | Variable |

# Who wins?

- No one wins, it’s all about context.
- Learn to use both.
- Use Proc SQL to simplify programs by combining several different tasks in one when you are dealing with small to medium size datasets.
- Use the datastep for large dataset processing with conditional statements and loops.
- Besides, no one really wants to see a car mechanic fight an old lady!

# References


- http://support.sas.com/documentation/cdl/en/lrcon/62955/HTML/default/viewer.htm#a000961108.htm
- http://support.sas.com/documentation/cdl/en/lrcon/68089/HTML/default/viewer.htm#p0e0mk25gs9binn1s9jiu4otau29.htm
- http://support.sas.com/documentation/cdl/en/lrcon/68089/HTML/default/viewer.htm#n1g8q3l1j2z1hjn1gj1hln0ci5gn.htm

# What I couldn’t cover but wish I did!

- Using multiple « set » or « merge » statements in the datastep.
- Joining data with the use of formats and hash tables.
- Working on several rows of data (through retains or lag statements).
- Using arrays.
- Views to allow efficient chain datastep processing.