PharmaSUG 2025 - Paper AP-121

SET Statement Considered Harmful

Bartosz Jabłoński, yabwon/Warsaw University of Technology

ABSTRACT

The SET statement is the primary way SAS® programmers access the data (I could bet $5, it is over 90% of cases). Since data processing in SAS is I/O oriented the majority of optimization can be done by reducing I/O, i.e., the proper use of the SET statement[1] in the code. The article is dedicated to beginner SAS programmers and considered education focused. In the paper we present a group of examples showing how (mis)use of the SET statement can be (and is) harmful for data processing and how the grim situation can be fixed.

Table of contents

WHEN SET DOES NOT MEAN "ALL SET"

Toutes choses sont dites déjà; mais comme personne n'écoute, il faut toujours recommencer.[2]

André Gide

Contrary to the famous article by Edgar Dijkstra (see [**Dijkstra 1968**]), in this text we are not going to discourage SAS programmers from using the SET statement. In fact, discouraging use of the SET statement could be rather hard (though not impossible) to achieve. In this article, we want bring beginner SAS programmer's attention to the following *good programming practice*, shared by Aster and Seidman in their book, to "*consider every SET statement[1] with suspicion*" (see [**Aster & Seidman 1996**]).

However strange as it might sound, a SAS data set (not a variable, not an array) is a fundamental data structure in SAS programming. All data communication is done through SAS data sets. Whenever we are placing data into the Program Data Vector (PDV) or pushing data off of it, we are using SAS data sets as a transport medium. Data sets by design are stored on disk drives and require input and output (I/O) operations to move data into and from memory. Even though modern disk drives went through huge technological transformation gaining a lot of speed, still I/O operations are one of the slowest (or rather most time-consuming) part of code execution. Thus, reducing the number of I/O operations seems to be the most natural way of improving program efficiency. If reading a data set one time takes some time, doing it twice doubles the time, etc. If one says: "*my data are small and read/write in fraction of a second*", we postulate that even small data, when read thousands of times, can become annoyingly "big".

[1] Of course also statements like: MERGE, INPUT, DATA=, etc.

[2] See "Conclusion" section for explanation

Of course, working with the SET statement is not the only way to "interact" with the data; reading in from or writing out to external text files, or processing through library engines, also fit into I/O operations realm, and because of that, can be considered potentially "harmful."

The "SET" statement in the article's title is a representative of a wider class of statements also containing: the MERGE, INPUT, DATA, DATA=, OUT=, or any other I/O related statement. In fact, the title could be: "*Use of too many input/output statements can be potentially harmful for a program performance*" (but you must admit, it would not be "it"). In fact, in the Dijkstra's article, it is not that the GOTO[3] statement is evil – it is the lack of structured programming that causes harm. Similarly, here, it is not the SET statement itself that is at fault . The thing we want to highlight is a general (so often haunting inexperienced SAS programmers) problem of writing code that in one or another way inefficiently uses I/O operations.

Being a beginner SAS programmer is not an easy job; that is why we want to help you in the journey by showing some of the most common I/O pitfalls and how to avoid them. To be clear, this article should *not* be considered exhaustive (frankly, we do not even dare to claim so). There is a rather high probability you will have to search through more literature to make yourself a better SAS programmer, but we hope you will be able to use this article as a convenient starting point or, at least, a "handy" support.

THE DATA

Let's assume we are starting with the following data set and text file:

code: data

```
options DLcreateDIR;
libname source "%sysfunc(pathname(WORK))/have";
filename source "%sysfunc(pathname(WORK))/have/have.txt";

data source.have;
  call streaminit(42);
  file source;
  do grp = 1 to 3333 by 2, 2 to 3333 by 2;
    length id $ 8;
    do id = "A","B","C","D","E","F","G","H","I","J","K","L","M",
            "N","O","P","Q","R","S","T","U","V","W","X","Y","Z";
      do number = 1 to rand("integer",17,42);
        obs+1;
        output;
        put grp id number obs;
      end;
    end;
  end;
run;
```

The data set HAVE located in the library named SOURCE, is rather moderate in size (somewhere around 80MB in volume). HAVE has the following variables: grp, id, number, and obs. Table 1 displays part of the data set. The LOG summarizes it as:

the log - observations and variables

```
NOTE: The data set SOURCE.HAVE has 2'555'488 observations and 4 variables.
```

The have.txt text file created contains the same data and is approximately 45MB in volume.

[3] Yes, the SAS language also has GOTO statements, both in the 4GL and the macro language.

Table 1: Data set SOURCE.HAVE, example

grp	id	number	obs	grp	id	number	obs	grp	id	number	obs	grp	id	number	obs
1	A	1	1	1	L	21	373	1	X	1	717	3	I	28	1090
1	A	30	30	1	M	1	374	1	X	18	734	3	J	1	1091
1	B	1	31	1	M	22	395	1	Y	1	735	3	J	24	1114
1	B	36	66	1	N	1	396	1	Y	26	760	3	K	1	1115
1	C	1	67	1	N	32	427	1	Z	1	761	3	K	27	1141
1	C	20	86	1	O	1	428	1	Z	37	797	3	L	1	1142
1	D	1	87	1	O	37	464	3	A	1	798	3	L	35	1176
1	D	36	122	1	P	1	465	3	A	20	817	3	M	1	1177
1	E	1	123	1	P	32	496	3	B	1	818	3	M	30	1206
1	E	35	157	1	Q	1	497	3	B	37	854	3	N	1	1207
1	F	1	158	1	Q	42	538	3	C	1	855	3	N	40	1246
1	F	40	197	1	R	1	539	3	C	37	891	3	O	1	1247
1	G	1	198	1	R	35	573	3	D	1	892	3	O	37	1283
1	G	34	231	1	S	1	574	3	D	17	908	3	P	1	1284
1	H	1	232	1	S	38	611	3	E	1	909	3	P	28	1311
1	H	18	249	1	T	1	612	3	E	42	950	3	Q	1	1312
1	I	1	250	1	T	37	648	3	F	1	951	3	Q	33	1344
1	I	42	291	1	U	1	649	3	F	40	990	3	R	1	1345
1	J	1	292	1	U	24	672	3	G	1	991	3	R	25	1369
1	J	22	313	1	V	1	673	3	G	33	1023	3	S	1	1370
1	K	1	314	1	V	17	689	3	H	1	1024	3	S	37	1406
1	K	39	352	1	W	1	690	3	H	39	1062	3	T	1	1407
1	L	1	353	1	W	27	716	3	I	1	1063	3	T	24	1430

THE LIST

Examples presented here have (more or less) the following form: first there is a short introduction, then (usually) two parallel snippets of code are compared, and some comments are shared. To be 100% clear, those examples and techniques used should not be perceived as "silver bullets", they are rather "80/20 rules" – i.e., *usually* working best practices. It might happen that your specific programming setup could be so exotic that you will basically have to use extra SET, MERGE, or similar statement, but that's life...

"TROLLING IS A ART"

Finding the following snippet in code you read you can be almost certain someone is playing a "troll."

code: the existing situation

```
/* a troll */
data source.have;
  set source.have;
run;
```

code: a smarter approach

```
/* solution */

/* a rare case of no-code/low-code */
/* beating programming             */
```

The DATA step is not only wasting I/O operations, your time, and space in the code file, but also can be potentially dangerous because overwriting may destroy potential indexes associated with the data set. And in case someone is really trying to get rid of an index, there are much better ways to do it, like PROC DATASETS, which we are going to talk about in a moment.

In this case we replace one data reading (I, input) and one data write (O, output) with zero, so the I/O savings can be symbolically presented as: $2\text{I/O} \longrightarrow 0\text{I/O}$, which gives an infinite (∞) gain.

"TWO BIRDS WITH ONE STONE"

When you need only a vertical subset (i.e., only selected variables) of your data for some sort of BY-group processing (i.e., sorting is required) you can do both in a single processing step.

code: the existing situation

```
/* two birds... */
data work.have;
  SET source.have;
  drop obs;
run;

proc sort data=work.have;
  by id grp;
run;
```

code: a smarter approach

```
/* ...one stone */
proc sort
  data=source.have(drop=obs)
  out=work.have
;
  by id grp;
run;
```

Instead of copying the sub-selected data and then sorting, we can do sorting directly on a subset of data. With this approach we replace two data reading (SET statement and DATA= option) and two data write (DATA statement and implicit output of the PROC SORT) with just two (DATA= and OUT= options), and the I/O reduction in this case is 4I/O$\longrightarrow 2$I/O.

[Note] This is a good moment to state a disclaimer. We are aware that such a simplified approach (i.e., presenting xI/O$\longrightarrow y$I/O reduction purely based on the number of SAS statements) and, especially, its precision might be debated, but the aim of the article is not to provide exact (up to every byte) I/O estimations. Our idea is to point out some programming styles which can result in better (or worse) performance of our programs.

"IT MAKES MY BLOOD BOIL"

I had a chance to see the following snippet multiple times when reviewing a coworker's code, and it always made (and still makes) my blood boil. It is almost like the "troll" but worse...

code: the existing situation

```
/* "blood boiler" */
data work.have;
  SET work.have;
  format number ROMAN12.;
run;
```

code: a smarter approach

```
/* "cooler" */
proc datasets lib=work noprint;
  modify have;
    format number ROMAN12.;
  run;
quit;
```

Instead of data read and write just to modify metadata, i.e., format assigning to the number variable, use of the PROC DATASETS makes much more sense. With the PROC DATASETS we can achieve our goal with just a fraction of the original I/O operations sacrifice. The bottom line is: 2I/O$\longrightarrow 0.1$I/O. We cannot write 0I/O because the PROC DATASETS has to touch metadata, what involves some I/O operations, but definitely much less than a "full" data step.

The Polish language idiom for "it makes my blood boil" is: "nóż się w kieszeni otwiera"[4] what literally translates to "a knife opens in one's pocket", and this reminds me that a very good introduction (and more) to PROC DATASETS can be found in [**Raithel 2010 & 2018**].

[4] See: https://en.wiktionary.org/wiki/n%C3%B3%C5%BC_si%C4%99_w_kieszeni_otwiera

"CUTTING, SLASHING, AND SHREDDING"

Sometimes we have to split data into separate data sets according to values of a variable, or according to a bunch of different splitting rules.

To split by values, people often involve macro-loops, but usually not the way it is "the way" for macro loops. Both snippets below, the left and the right one, can be imagined as a result of executing %DO-LOOP(s) over values: 1, 2, and 3.

code: the existing situation

```
/* silly */
data work.group_1;
  SET source.have;
  where grp=1;
run;
data work.group_2;
  SET source.have;
  where grp=2;
run;
data work.group_3;
  SET source.have;
  where grp=3;
run;
```

code: a smarter approach

```
/* smarter */
data work.group_1
     work.group_2
     work.group_3;
  SET source.have;
  where grp in (1 2 3);
  select(grp);
    when(1) output work.group_1;
    when(2) output work.group_2;
    when(3) output work.group_3;
    otherwise;
  end;
run;
```

Though both achieve the same final effect, the processing efficiency differs. The left one just wraps a 4GL code snippet of the whole DATA step inside the loop's body, like this:

code: 1 loop

```
%do i = 1 %to 3;
  data work.group_&i.;
    SET source.have;
    where grp=&i.;
  run;
%end;
```

When we think twice, and realize (or rather remind ourselves) that macro language is not a *4GL-code* generator but, it is a *whatever-text-you-want* generator(!), we can write it the "right" way, and easily reduce not needed I/O operations with three macro loops:

code: 3 loops

```
data %do i = 1 %to 3;
     work.group_&i.
     %end;;
  SET source.have;
  where grp in (%do i = 1 %to 3; &i. %end;);
  select(grp);
    %do i = 1 %to 3;
      when(&i.) output work.group_&i.;
    %end;
    otherwise;
  end;
run;
```

We get the same three data writes, but only one data read!

The trick with splitting data in a single read works well also for the "different splitting rules" case, even if result data sets have "overlapping" observations. In such cases, instead SELECT statement, a "good-old" IF-THEN conditional logic does the job:

code: the existing situation

```
data work.group_1;
  SET source.have;
  where grp in (1 2 3331 3332);
run;

data work.group_2;
  SET source.have;
  where "A"<id<"C" or "X"<id<"Z";
run;
```

code: a smarter approach

```
data work.group_1 work.group_2;
  SET source.have;

  if grp in (1 2 3331 3332) then
    output work.group_1;
  if "A"<id<"C" or "X"<id<"Z" then
    output work.group_2;
run;
```

In the scenarios we have considered, reduction went, respectively, from 6I/O to 4I/O, and from 4I/O to 3I/O, but in the general case it goes: $2n$I/O$\longrightarrow n+1$I/O, where n is the number of "groups/rules" we have to split the data.

"THE FINAL (BACKWARD) COUNTDOWN"

Sometimes we may be faced with a task that requires reversing our data order, i.e. reading a data set from bottom to top. In such a situation, instead of creating an artificial ordering variable n and resorting our newly created data by it, it is much better (and cheaper) to use the POINT= option in the SET statement.

code: the existing situation

```
/* sloppy */
data work.have;
  SET source.have;
  n + 1;
run;

proc sort data=work.have
          out=work.have(drop=n);
  by descending n;
run;
```

code: a smarter approach

```
/* smarter */
data work.have;
  do point = nobs to 1 by -1;
    SET source.have point=point
                    nobs=nobs;
    output;
  end;
stop;
run;
```

Explicit DO-LOOP reads the data backward and gives the following reduction: 4I/O$\longrightarrow 2$I/O.

"DISENCHANTING"

Sometimes people may think that reading data from an external file with help of INFILE and INPUT statements requires "special" treatment. In fact there is nothing special in such DATA steps.

code: the existing situation

```
data work.have;
  infile source;
  INPUT grp id $ number obs;
run;

data work.have2;
  SET work.have;
  if grp NE 1;
  number2 = number + 1000;
run;
```

code: a smarter approach

```
data work.have2;
  infile source;
  INPUT grp id $ number obs;
  if grp NE 1;
  number2 = number + 1000;
run;
```

There is no need to split our processing in two steps. We can read data directly from an external file, filter them out, derive new variables, or do calculations. Again: 4I/O$\longrightarrow 2$I/O.

"DISENCHANTING" AGAIN

The aim of this tip is also to "demystify", this time the MERGE statement. For this example let's assume that we were provided with those two data sets: work.data1 (2457997 observations) and work.data2 (2554654 observations). Our goal is, after merging those two data sets over variable obs, to aggregate variables number2 and number3

code: 1st data set

```
data work.data1;
  SET source.have;
  where id ne "B";
  number2=number*number;
run;
```

code: 2nd data set

```
data work.data2;
  SET source.have;
  where grp ne 17;
  number3=number + 17;
run;
```

One could think that merging two data sets has to be executed exclusively, so that in the next step we could filter, summarize or use options like for example END=. Nothing could be further from the truth than that. The MERGE statement accepts: END=, NOBS=, or INDSNAME= options, same way the SET statement does. Furthermore, there is nothing wrong in not generating a data set (i.e. data _null_;) from a DATA step that merges data.

code: the existing situation

```
/* wandering around */
data work.interimStep;
  MERGE work.data1 work.data2;
  by obs;
run;
data OneObs;
  set work.interimStep END=_E_;
  sum + (number2 + number3);
  if _E_;
  put sum;
run;
```

code: a smarter approach

```
/* straight to the point */
data _null_;
  MERGE work.data1 work.data2 END=_E_;
  by obs;
  sum + (number2 + number3);
  if _E_;
  put sum;
run;
```

This way we gain some I/O savings too, 4.01I/O$\longrightarrow 2$I/O (the 0.01 is for the OneObs).

"DISENCHANTING" A BIT MORE

We have taken already away a bit of "holiness" from the MERGE statement, and we are not going to stop there. After playing a bit with SQL, a beginner SAS user might think that similarly to SQL's join the MERGE allows only two tables. And that is not true. The MERGE statement allows us to combine data from as many data sets as we want, of course as long as the merging is done over the same variable.

code: the existing situation

```
/* I/O waste */
data work.step1;
  MERGE source.have work.data1;
  by obs;
run;

data work.final;
  MERGE work.step1 work.data2;
  by obs;
run;
```

code: a smarter approach

```
/* smarter */
data work.final;
  MERGE source.have
        work.data1
        work.data2;
  by obs;
run;
```

So, instead of executing several steps with two data sets each we can simply do one with all data sets merged at once. This way we again gain some more I/O savings, 6I/O$\longrightarrow 4$I/O, and in general case $3n$I/O$\longrightarrow n+2$I/O, in this case n is the number of data sets to merge, not counting the "main" data set.

"NICE PIECE OF ABSOLUTELY USELESS STEP"

Sometimes when we are focused on deriving next and next variables we may fall into a trap called by Aster and Seidman in [**Aster & Seidman 1996**] a "stream of consciousness" programming: one thought (derivation) one DATA step. Instead of producing a bunch of intermediate DATA steps, and through those DATA steps more data sets, and eventually re-merging those results (fortunately now we know only one MERGE data step is enough!) we can sometimes reduce I/O operations and combine processing into one data step.

code: the existing situation

```
/* stream of consciousness */
data work.step1;
  SET source.have;
  where number > 20;
  number2=number*number;
  keep obs number2;
run;
data work.step2;
  SET source.have;
  where id NE "B";
  number3=number+17;
  keep obs number3;
run;
/*data work.step3;
    SET source.have;
    where id > "X";
    number4=number**42;
    keep obs number4;
  run;*/
data work.final;
  MERGE source.have
        work.step1
        work.step2
      /*work.step3*/
  ;
  by obs;
run;
```

code: a smarter approach

```
/* smart */
data work.final;
  SET source.have;

  if number > 20 then
    number2=number*number;

  if id NE "B" then
    number3=number+17;
  /*
  if id > "X" then
    number4=number**42;
  */
run;
```

Sometimes such code has one more advantage: if we have to comment out a part of the code (e.g., a variable derivation or a conditional processing) we only need to comment out in one place of the code. Savings here are 8I/O$\longrightarrow 2$I/O, but the I/O reduction as a function of the derivations number n would have the following form $2+3n$I/O$\longrightarrow 2$I/O.

"DOING MORE BY DOING LESS"

For this example, let's assume that we were provided with those two data sets: work.one (842279 observations) and work.two (2457997 observations). This time our goal is to create three data sets: a) the first with only those OBS values from WORK.ONE that does not show up in WORK.TWO, b) the second that has observations with a reversed relation, and c) the third that contains OBS values in the intersection of WORK.ONE and WORK.TWO.

code: 1st data set

```
data work.one;
  set source.have;
  where number > 20;
run;
```

code: 2nd data set

```
data work.two;
  infile source;
  INPUT grp id $ number obs;
  if id ne "B";
run;
```

Classic PROC SQL approach can be replaced with shorter and clearer MERGE data step. Of course, this example quietly assumes that WORK.ONE and WORK.TWO are sorted by values of OBS.

code: the existing situation

```
/* "classic" */
proc sql;
 create table only_in_one as
  select one.*
  FROM work.one
  left join
       work.two
  on one.obs = two.obs
  where two.obs is missing;
 create table only_in_two as
  select two.*
  FROM work.one
  right join
       work.two
  on one.obs = two.obs
  where one.obs is missing;
 create table one_and_two as
  select one.*
  FROM work.one
  inner join
       work.two
  on one.obs = two.obs;
quit;
```

code: a smarter approach

```
/* smarter */
data
  only_in_one
  only_in_two
  one_and_two;

  MERGE work.one(in=o1)
        work.two(in=t2);
  by obs;

  select;
    when(    o1 and not t2)
      output only_in_one;
    when(not o1 and     t2)
      output only_in_two;
    when(    o1 and     t2)
      output one_and_two;
    otherwise;
  end;
run;
```

The reduction here, $9\mathtt{I/O} \longrightarrow 5\mathtt{I/O}$, may not look so spectacular, but again we can see the superiority of the MERGE statement skills (of course, as long as we are staying in the world of $1\text{–}to\text{–}1$ and $1\text{–}to\text{–}N$ merges).

"DOING BY NOT DOING"

Imagine you have two (in practice more) data sets (for simplicity, with excluding data) produced by the following snippet:

code: data

```
data work.one work.two;
  set source.have;
  if id NE "C" then output work.one;
               else output work.two;
run;
```

For example, one data set contains data for placebo subjects and the other for treatment. Or maybe the first data set has data from January, the second from February, the third from March, etc.

Our task is to concatenate data (i.e. to stack data from the second data sets behind data from the first, and so on) so we can read them sequentially. There is a non-zero probability that a DATA step with sequential SET statement reading both data sets and overwriting the first one would be the first choice. But for such a job PROC APPEND, moving only data from the second data set could work better (giving reduction $3\mathtt{I/O} \longrightarrow 1\mathtt{I/O}$, as the BASE= data set is not touched)

code: the existing situation

```
/* silly */
data work.one;
  SET work.one work.two;
run;
```

code: a smarter approach

```
/* smarter */
proc append base=work.one
            data=work.two;
run;
```

But since we are going to read the data in the subsequent step, maybe we do not have to move data at all?

By creating SAS view we can reduce I/O operations of the concatenation process to zero and materialize it only when we really need it, just like in this snippet:

code: a smarter approach

```
/* possibly even smarter */
data work.onetwo / view=work.onetwo;
  SET work.one work.two;
run;
```

"NO RAW MACRO LOOPS", PART 1.

The macro language is a very practical way to write reusable code. Wrapping a snippet in a macro and have it for every call seems like a good idea, but sometimes, especially with the wrong application of %DO-LOOPs, it may slow your processing down (we have seen examples of this earlier). In his blog posts Rick Wicklin wrote about doing things "the BY way" multiple times (see [**Wicklin 2012**] or [**Wicklin 2017**]). The following examples will present work driven by a spirit similar to the one described by Rick, i.e., reducing unnecessary macro loops.

The first example is pretty easy to follow. Let's assume we have to calculate percentiles for our data in groups over variable id. A naive (but not so uncommon) approach would be to write a macro that extracts the list of id values and then execute a %DO-LOOP over the list. The loop, in each iteration, subsets the data set, then runs the PROC UNIVARIATE to calculate percentiles, and eventually use the PROC APPEND to combine results. In this, or any other, approach the PROC UNIVARIATE seems to be rather unavoidable, especially if we consider how very robust it is for calculating percentiles. But all those sub-setting DATA steps and all that appending can be avoided.

code: the existing situation

```
%macro slow();
  proc sql;
    select distinct id
    into :id_list separated by " "
    FROM source.have;
    %let n = &SQLobs.;
  quit;
  %do i = 1 %to &n.;
    %let id = %scan(&id_list.,&i.);
    data subset;
      SET source.have;
      where id = "&id.";
    run;
    proc univariate data=subset;
      var number;
      output out=p
        pctlpre=P pctlpts=0 to 100;
    run;
    proc append base=pctl1 data=p;
    run;
  %end;
%mend slow;
%slow()
```

code: a smarter approach

```
/* smarter */
proc univariate data=source.have;
  class id;
  var number;
  output out=pctl2
    pctlpre=p pctlpts=0 to 100;
run;
```

Instead of macro looping we can do processing in groups, right? You could argue: "*But to do the BY-group processing we still have to sort the data by the grouping variable*". Well, that is true - but fortunately PROC UNIVARIATE provides the CLASS statement that saves our day. It is important to remember that the CLASS statement can specify at most two variables to be used to group the data into classification levels!

The I/O reduction here has the following form: $2+5n$I/O$\longrightarrow 2$I/O, the 2 on the left side is for SQL and for the first creation of pctl1 data set. There is one more advantage of the CLASS approach, it not only reduces I/O operations, but also gives us values of the id variable in the final pctl2 data set.

"NO RAW MACRO LOOPS", PART 2.

The second example is a bit more complicated. It is based on a true story, though any similarities...

Assuming we have an input data set have grouped by variable grp, though not necessarily sorted, a dictionary dict of separate parameters for each value of grp variable, and we want to execute a parameterized calculation of a new variable, let's say something like the following:

$$x = \texttt{function}(\alpha_{grp} \times var_1, \beta_{grp} \times var_2, \gamma_{grp} \times var_3) + Constant_{grp};$$

where α, β, γ, and $Constant$ are group dependent, i.e., their values changes depending on the grp variable values (that is why we are picking those values from the dictionary). And the function$(\ldots)$ can be considered as a group of conditional logic expressions composed of SAS statements and SAS functions (including PROC FCMP user defined functions too).

In our example, we are going to use the RAND( ) function and the following snippet:

code: formula to evaluate

```
x = rand("Distribution", ParA, ParB) + Threshold;
```

The RAND( ) function accepts various number of parameters - depending on the distribution used - so it perfectly fits into our need for a "pretender-of-some-business-logic", by being semantically simple enough and syntactically complicated enough at the same time. The dictionary is a SAS data set source.dict with the following (long) structure, and for brevity we have it prepared only for selected values of id variable from the source.have. Here is the code for the dictionary:

code: dictionary

```
data source.dict;
  array distr[9] $ 4 ("BERN" "CAUC" "EXPO" "F" "GAMM" "INTE" "LOGI" "T" "UNIF");
  array Npar[9]  $ 4 ("1"    "0"    "1"    "2" "1"    "2"    "2"    "1" "2"   );
  array ParA[9]  $ 4 ("0.5"  " "    "1"    "3" "42"   "0"    "0"    "1" "0"   );
  array ParB[9]  $ 4 (" "    " "    " "    "4" " "    "10"   "1"    " " "1"   );

  do i = 1 to 9;
    length id $ 8;

    id = char(distr[i],1);
    par="Distribution";
    val=distr[i]; output;
    par="Npar";
    val=Npar[i]; output;
    par="ParA";
    val=ParA[i];
    if val NE "" then output;
    par="ParB";
    val=ParB[i];
    if val NE "" then output;
    par="Threshold";
    val = put(i/10,3.1); output;
  end;
  keep id par val;
run;
```

and here is the output of the dictionary displayed in two columns tabular form:

Table 2: Data set SOURCE.DICT

id	par	val
B	Distribution	BERN
B	Npar	1
B	ParA	0.5
B	Threshold	0.1
C	Distribution	CAUC
C	Npar	0
C	Threshold	0.2
E	Distribution	EXPO
E	Npar	1
E	ParA	1
E	Threshold	0.3
F	Distribution	F
F	Npar	2
F	ParA	3
F	ParB	4
F	Threshold	0.4
G	Distribution	GAMM
G	Npar	1
G	ParA	42
G	Threshold	0.5
I	Distribution	INTE
I	Npar	2
I	ParA	0
I	ParB	10
I	Threshold	0.6
L	Distribution	LOGI
L	Npar	2
L	ParA	0
L	ParB	1
L	Threshold	0.7
T	Distribution	T
T	Npar	1
T	ParA	1
T	Threshold	0.8
U	Distribution	UNIF
U	Npar	2
U	ParA	0
U	ParB	1
U	Threshold	0.9

The solution I witnessed (left column below) was based on the following logic:

(1) start the process and create a list of unique values of the `id` variable from the dictionary [lines 2-8],
(2) loop over the list and for *each* value from the list [lines 10,11,28]:
  (a) sub-set the dictionary for given `id` value and produce a bunch of macro variables named like parameters and with their corresponding values as values [lines 12-16],
  (b) sub-set the `have` data set for given `id` value and for every given set of macro variables execute calculation [lines 17-24],
  (c) stack the result by appending it to previous iteration's result [lines 25-27]
(3) end the process [line 29].

The number of `I/O` operations here is $5n$`I/O`, where n is the number of the `id` variable values (9 in our case), plus one `I/O` for SQL and one for the first execution of the `PROC APPEND`. So, in the example here the number of inputs and outputs ends to be 47`I/O`.

code: the existing situation

```
/* slow */
%macro slow2();
 proc sql;
  select distinct id
  into :id_list separated by " "
  FROM source.dict;
  %let n = &SQLobs.;
 quit;

 %do i = 1 %to &n.;
  %let id = %scan(&id_list.,&i.);
  data dict;
   SET source.dict;
   where id = "&id.";
   call symputX(par,val,"L");
  run;
  data tmp;
   SET source.have;
   where id = "&id.";
   x = rand("&Distribution."
   %if &Npar.>0 %then %do;,&ParA.%end;
   %if &Npar.>1 %then %do;,&ParB.%end;
   ) + &Threshold.;
  run;
  proc append base=work.result1
              data=tmp;
  run;
 %end;
%mend slow2;

options Mprint;
%slow2()
```

code: a smarter approach

```
/* smarter */
proc transpose data=source.dict
               out=dict(drop=_:);
  by id;
  id par;
  var val;
run;

PROC SQL;
  create table work.result2 as
  select h.*,
    case
      when d.Npar="2" then
        rand(Distribution
            ,input(ParA, best.)
            ,input(ParB, best.))
      when d.Npar="1" then
        rand(Distribution
            ,input(ParA, best.))
      else
        rand(Distribution)
    end + input(Threshold, best.) as x
  from
    source.have as h
    inner join
    dict as d
  on h.id = d.id;
QUIT;
```

An alternative to the first approach is based only on 5I/O, the first two are for the PROC TRANSPOSE which transforms the dictionary from long into wide format (see table printout below). The other three are for the PROC SQL where the CASE-WHEN-END conditional expression combined with several calls to the INPUT() function does the heavy lifting. Remembering that n is the number of the id variable values we eventually end up with $2{+}5n$I/O$\longrightarrow 5$I/O reduction.

Table 3: Data set with dictionary after transposition

id	Distribution	Npar	ParA	ParB	Threshold
B	BERN	1	0.5		0.1
C	CAUC	0			0.2
E	EXPO	1	1		0.3
F	F	2	3	4	0.4
G	GAMM	1	42		0.5

id	Distribution	Npar	ParA	ParB	Threshold
I	INTE	2	0	10	0.6
L	LOGI	2	0	1	0.7
T	T	1	1		0.8
U	UNIF	2	0	1	0.9

"#HASH TABLE FOR HELP"

Classical problem of re-merging summary statistics back with the original data (often seen as a note after PROC SQL execution) usually is done by a) sorting data by grouping variables, b) doing some BY-group processing to calculate statistics, and c) merging created summary statistics data set with the source data.

code: the existing situation

```
/* standard */
proc sort data=source.have
          out=work.have;
  by id;
run;
data work.aggr;
  SET work.have;
  by id;
  if first.id then
    do;
      maxN=number;
      minN=number;
    end;
  maxN = maxN max number;
  minN = minN min number;
  if last.id then
    do;
      range = maxN-minN;
      output;
    end;
  keep id range;
  retain maxN minN;
run;
data work.want;
  merge work.have work.aggr;
  by id;
  shift = number/range;
  drop range;
run;
```

code: a smarter approach

```
/* with hash tables */
data work.want;
 dcl hash S(ordered:"A");
 S.defineKey("id");
 S.defineData("id","maxN","minN");
 S.defineDone();
 declare hiter I("S");

 dcl hash D(multidata:"Y",ordered:"A");
 D.defineKey("id");
 D.defineData("grp","id","number","obs");
 D.defineDone();

 do until(_E_);
  SET source.have END=_E_;
  D.add();

  shift = S.find();
  maxN = max(maxN,number);
  minN = min(minN,number);
  S.replace();
 end;

 do while(0=I.next());
  do while(D.do_over()=0);
   shift = number/(maxN-minN);
   output;
  end;
 end;
stop;
drop maxN minN;
run;
```

Use of hash tables can give us pretty nice I/O operations reduction if we use them both for storing summary statistics and the source data. Like in the example above hash table S stores minimum and maximum for id groups, and hash table D stores all data from the input data set. Use of the explicit DO-UNTIL loop allows us to read-in source data and calculate statistics at the same time with only one data pass. Then in a double DO-WHILE loop we combine data and the aggregate to produce final result. Original seven I/O operations reduce to two giving: $7\text{I/O} \longrightarrow 2\text{I/O}$. Of course, we realize that reduction of I/O operations here has a price of higher memory consumption. But often that price is worth paying, especially that hash tables allow us to reduce or stop I/O intensive data sorting. The best hash tables source of knowledge is [**Dorfman & Henderson 2018**] book.

"NO RAW MACRO LOOPS", PART 2. - REVISITED

If consider our just gained hash tables experience and we add some "good old" arrays to it, we can go even one step further with I/O operations reduction.

code: hash tables approach

```
data work.result3;
  if 1=_N_ then do;
    declare hash D();                  declare hash I();
    D.defineKey("id", "par");          I.defineKey("id");
    D.defineData("val");
    D.defineDone();                    I.defineDone();
    do until(_E_);
      SET source.dict end=_E_; /* one data reading for 2 hash tables */
      if D.add() then stop;    /* "quality stop" if dict has doubles */
      I.replace();
    end;
  end;

  SET source.have;
  by ID notsorted;
  if 0=I.check(); /* to get "inner join" effect */

  if first.ID then do; /* get parameters */
      array Dict[*] $ Distribution Npar ParA ParB Threshold; retain;
      call missing(of Dict[*]);
      do j=1 to Dim(Dict);
        if 0=D.find(key:id,key:vname(Dict[j])) then Dict[j] = val;
      end;
    end;
  select(Npar);
    when("2") x = rand(Distribution,input(ParA, best.),input(ParB, best.));
    when("1") x = rand(Distribution,input(ParA, best.));
    otherwise x = rand(Distribution);
  end;
  x = x + input(Threshold, best.);

  drop par val j Distribution Npar ParA ParB Threshold;
run;
```

We end up with $2+5n$I/O$\longrightarrow 3$I/O instead of 5I/O. But this example cannot be left without a comment. There is a fair chance that the PROC TRANSPOSE + PROC SQL's approach, though it has two more I/O operations, will win here (in terms of "wall clock" time) especially when the dictionary is small. This is because PROC SQL is designed smart and if it figures out the dictionary table is small enough to fit in-memory, it will use so called "hash join" method to combine data faster. If you run PROC SQL from our example with (undocumented) option: _METHOD[5], it will print something similar to this:

the log - SQL and _method

```
NOTE: SQL execution methods chosen are:
      sqxcrta /* Create table */
          sqxjhsh /* Hash join operation */
              sqxsrc( SOURCE.HAVE(alias = H) ) /* Source rows from table */
              sqxfil /* Rows filtration */
                  sqxsrc( WORK.DICT(alias = D) )
```

[5] See [**Lavery 2005**] to learn the _method secrets

Of course, if the program logic requires data step solutions, e.g., arrays, this one is clearly the winner. The example also shows that: a single data read can populate multiple hash tables, what is an extremely convenient trick. Further more we did not have to sort the `source.have` data set to do the table join. The bottom line here is that, in general, hash tables give us a bunch of useful I/O-saving features.

"ONE STEP TO RULE THEM ALL"

Sometimes there is a need to create a "template" data set that contains all possible combinations of categorical variables. A "standard" PROC SQL approach, of creating a few data sets with unique values, crowned by SQL's Cartesian product of those, is often the choice. But, in such a case, the much more I/O-efficient is a data step.

code: SQL 1

```
proc sql;
create table
  work.sql_dist_number as
select
  distinct number
from
  source.have;

create table
  work.sql_dist_id as
select
  distinct id
from
  source.have;

create table
  work.sql_dist_grp as
select
  distinct grp
from
  source.have;

create table
  all_possible_crosses_S as
select *
from
  work.sql_dist_number,
  work.sql_dist_id,
  work.sql_dist_grp;
quit;
```

code: hash table

```
data all_possible_crosses_H;
  declare hash H1(ordered:"A");
  H1.defineKey("number");
  H1.defineDone();
  declare hiter i1("H1");

  declare hash H2(ordered:"A");
  H2.defineKey("id");
  H2.defineDone();
  declare hiter i2("H2");

  declare hash H3(ordered:"A");
  H3.defineKey("grp");
  H3.defineDone();
  declare hiter i3("H3");

  do until(_E_);
    set source.have(keep=number id grp)
        end=_E_;
    H1.ref();
    H2.ref();
    H3.ref();
  end;

  do while(0=i1.next());
    do while(0=i2.next());
      do while(0=i3.next());
        output;
      end;
    end;
  end;
stop;
run;
```

Use of hash tables to store unique values allows us to traverse data only once and the result is produced directly. The I/O reduction, in the example here, goes from 10I/O (read, write, and again read - for each variable, plus one for result data set) to just 2I/O (one for reading `source.have` and the second for writing the result data set). In general case, having n variables selected, it goes $1+3n$I/O$\longrightarrow 2$I/O.

An inquisitive reader may doubt the efficiency, and suspect we are pulling the wool over one's eyes, and even support the doubts with the following (shorter) SQL query that seems to reduce part of those I/O operations:

code: SQL 2

```
proc sql _method;
  create table all_possible_cross_S2 as
  select *
  from (select distinct number from source.have)
     , (select distinct id     from source.have)
     , (select distinct grp    from source.have)
  ;
quit;
```

but, when the execution is closely inspected with the _METHOD option, we can see the source.have data set is read 3 times, and the Cartesian join (sqxjsl) needs some utility storage too. We have at least four evident I/O operations (including write of the result data set) and some unknown number of "under the hood" utility data wranglings.

the log - SQL and _method

```
NOTE: SQL execution methods chosen are:
      sqxcrta
          sqxjsl /* Step loop join (Cartesian) */
              sqxunqs
                  sqxsrc( SOURCE.HAVE )
              sqxjsl
                  sqxunqs /* Select unique values */
                      sqxsrc( SOURCE.HAVE )
                  sqxunqs
                      sqxsrc( SOURCE.HAVE )
NOTE: SAS threaded sort was used.
```

"I HAVE SEEN THIS BEFORE"

Yet another example of an "I-have-seen-this-before" situation is the case where we have to add (combine) some new data into our base table, furthermore:

(1) the data are provided in separate data sets (no Excels, CSVs, etc., just for simplicity),
(2) the data we want to combine must be merged over different variables, and
(3) the final data set should maintain the original observations order.

For simplicity, data sets with additional data have only two variables each, one for "joining key" and the other for data, e.g., number and number_text, etc.

Usually, two approaches are used to solve the task. The first one combines merging and sorting intertwined together, the second approach, as one can expect, is SQL's left joining.

The "merge and sort" solution starts with ordering all additional data sets by their merging variables. Next, the base data set is sorted by selected variable and then merged with additional data. The process is repeated for all variables, and eventually the final data set is resorted to restore the original observations order. In our example setup the process involves 23 I/O. The alternative, with PROC SQL, seems to be less I/O intense, at least at the first glance. Unfortunately, the _METHOD option reveals it is not just a '94 "Four Reads and a Write" comedy but it is (again) much more complicated. The LOG shows that before joining, all data sets are resorted, and since the merge join is used, the temporary data have to be stored in utility files. Of course the final re-sorting takes its toll too.

code: Merge and sort

```
proc sort
  data = number_data
  out = number_data_sort;
  by number;
run;
proc sort
  data = id_data
  out = id_data_sort;
  by id;
run;
proc sort
  data = grp_data
  out = grp_data_sort;
  by grp;
run;
proc sort
  data = source.have
  out = have_by_grp;
  by grp;
run;
data have_1;
  merge
    have_by_grp
    grp_data_sort;
  by grp;
run;
proc sort
  data = have_1
  out = have_by_id;
  by id;
run;
data have_2;
  merge
    have_by_id
    id_data_sort;
  by id;
run;
proc sort
  data = have_2
  out = have_by_number;
  by number;
run;
data have_3;
  merge
    have_by_number
    number_data_sort;
  by number;
run;
proc sort
  data = have_3
  out = combined_DS;
  by obs;
run;
```

code: SQL

```
proc sql _method;
  create table combined_SQL as
  select
    h.*,
    n.number_text,
    i.id_text,
    g.grp_text
  from
    source.have as h
  left join
    number_data as n
    on h.number=n.number

  left join
    id_data as i
    on h.id=i.id

  left join
    grp_data as g
    on h.grp=g.grp

  order by obs;
quit;
```

The LOG unveils it all. Even if we assume that the `sqxsrc` and `sqxsort` utilize `I/O` operations together we can count something around eleven `I/O` in the process.

the log - SQL and _method

```
NOTE: SQL execution methods chosen are:
      sqxcrta
          sqxsort
              sqxjm
                  sqxsort
                      sqxsrc( WORK.GRP_DATA(alias = G) )
                  sqxsort
                      sqxjm
                          sqxsort
                              sqxsrc( WORK.ID_DATA(alias = I) )
                          sqxsort
                              sqxjm /* Merge join operation */
                                  sqxsort
                                      sqxsrc( WORK.NUMBER_DATA(alias = N) )
                                  sqxsort /* Sort operation */
                                      sqxsrc( SOURCE.HAVE(alias = H) )
NOTE: SAS threaded sort was used.
```

This time, with a little help from our hash table friends, the "optimized" solution proposed here uses only 5I/O: three data reads to populate hash tables: H1, H2, and H3 with data, one run over the source.have base data set, and eventually for writing the final combined_hash data set, with no need for any re-sorting at all (which also reduces Operating System Memory cost by 15 times!). The I/O costs reduction goes 23I/O$\longrightarrow 5$I/O. And just to be clear, code in line number 2 does not read any data during the execution phase. The SET statement is there just for metadata extraction in the compilation phase and is used just to minimize variables metadata interaction (simple LENGTH would work too).

code: hash tables approach

```
1  data combined_hash;
2    if 0 then set work.number_data work.id_data work.grp_data;
3
4    declare hash H1(dataset:"work.number_data");
5    H1.defineKey("number");
6    H1.defineData("number_text");
7    H1.defineDone();
8
9    declare hash H2(dataset:"work.id_data");
10   H2.defineKey("id");
11   H2.defineData("id_text");
12   H2.defineDone();
13
14   declare hash H3(dataset:"work.grp_data");
15   H3.defineKey("grp");
16   H3.defineData("grp_text");
17   H3.defineDone();
18
19   do until(_E_);
20     set source.have
21         end=_E_;
22     if H1.find() then number_text="";
23     if H2.find() then id_text="";
24     if H3.find() then grp_text="";
25     output;
26   end;
27 stop;
28 run;
```

CONCLUSION

In this article we tried to bring beginner SAS programmers' attention to mastering I/O operations efficiency by "pruning" redundant SET, MERGE, INPUT, or any other I/O related statements. Undoubtedly, we can state that SAS is a very syntax rich and flexible language, and it takes quite some effort to master it. Even tough examples and snippets presented here may seem to be "repetitions of obvious things", like the 1947 Nobel Prize in Literature winner, French, André Gide[6] said:

"*Toutes choses sont dites déjà; mais comme personne n'écoute, il faut toujours recommencer.*"

("*Everything has been said before, but since nobody listens we have to keep going back and beginning all over again.*")

From time to time, new, inexperienced SAS programmers show up in the SAS community seeking ways to make their code better, thus we strongly believe that some of those "seems to be obvious" programming rules should be presented again and again. Writing good and efficient code is a non-trivial task. The article, by use of the inevitable part of the education process - repetition, tries to help to get to the stage when "we know how to write good code" easier, without need to first wander around through meanders and off-roads of ugly or (probably more often) inefficient code. We hope that the topics presented in the article will help to make life of "junior" SAS programmers easier and allow them to shorten the "from bad to good code" path they are walking. Hopefully, programmers with more experience will also find the paper useful.

Thus, to make your program efficient, let's repeat after [**Aster & Seidman 1996**] one more time: "*consider every SET (MERGE, INPUT, and any other I/O related) statement with suspicion*!"

The End

REFERENCES

[Dijkstra 1968] Edsger Dijkstra, "Go To Statement Considered Harmful", Communications of the ACM. 11 (3): 147-148, 1968, https://doi.org/10.1145%2F362929.362947, https://homepages.cwi.nl/~storm/teaching/reader/Dijkstra68.pdf

[Aster & Seidman 1996] Rick Aster, Rhena Seidman, "Professional SAS Programming Secrets", McGraw-Hill Inc., US; 2nd Updated Edition, 1996

[Lavery 2005] Russ Lavery, "The SQL Optimizer Project: _Method and _Tree in SAS 9.1", SUGI 30 Proceedings, 2005, https://support.sas.com/resources/papers/proceedings/proceedings/sugi30/101-30.pdf

[Raithel 2010 & 2018] Michael A. Raithel, "PROC DATASETS; The Swiss Army Knife of SAS Procedures", WUSS Proceedings, 2018, https://www.lexjansen.com/wuss/2018/144_Final_Paper_PDF.pdf SGF Proceedings, 2010, https://support.sas.com/resources/papers/proceedings10/138-2010.pdf

[Wicklin 2012] Rick Wicklin, The DO-LOOP blog, 2012 "Simulation in SAS: The slow way or the BY way", https://blogs.sas.com/content/iml/2012/07/18/simulation-in-sas-the-slow-way-or-the-by-way.html

[Wicklin 2017] Rick Wicklin, The DO-LOOP blog, 2017 "An easy way to run thousands of regressions in SAS", https://blogs.sas.com/content/iml/2017/02/13/run-1000-regressions.html

[Dorfman & Henderson 2018] Paul M. Dorfman, Don Henderson, "Data Management Solutions Using SAS Hash Table Operations: A Business Intelligence Case Study", SAS Institute Press, 2018

ACKNOWLEDGMENTS

We would like to thank Filip Kulon, Troy Martin Hughes, and Louise Hadden! We are utterly grateful for their help in "polishing" this text.

[6] Quote taken from: https://en.wikiquote.org/wiki/Andr%C3%A9_Gide

CONTACT INFORMATION

Your comments and questions are valued and encouraged!

Contact Bart at one of the following e-mail addresses:

yabwon✉gmail.com or bartosz.jablonski✉pw.edu.pl

or via the following LinkedIn profile: www.linkedin.com/in/yabwon or at the communities.sas.com by mentioning @yabwon.

Appendix A - code coloring guide

The best experience for reading this article is in color and the following convention is used:

- The code snippets use the following coloring convention:

code: is surrounded by a black frame

```
In general we use black ink for the code but:
- for reading clarity we sometimes mark code in orange ink,
- and comments pertaining to code are in a bluish ink for easier reading.
```

- The LOG uses the following coloring convention:

the log - is surrounded by a blueish frame

```
The source code and general log text are blueish.
Log NOTEs are green.
Log WARNINGs are violet.
Log ERRORs are red.
Log text generated by the user is purple.
```

INDEX