PharmaSUG 2025 - Paper HT-115

Route Sixty-Six to SAS® Programming*,

or 63 (+3) Syntax Snippets for a Table Look-up Task, or How to Learn SAS by Solving Only One Exercise!

Bartosz Jablonski - yabwon / Warsaw University of Technology
Quentin McMullen - Siemens Healthineers

ABSTRACT

It is said that a good programmer should be lazy. And what about a good programming teacher? We dare to say the same is true. This article will show that you can be lazy and also be able to teach an entire SAS®course at the same time!

The aim of the article is to present a variety of examples of how to do one of the most common data processing programming tasks, table look-up, in SAS. We don't assess these methods from a benchmarking or performance perspective but rather present them as an intellectual puzzle. Our goal is to explore how much SAS syntax (statements, PROCs, functions, etc.) could be taught using only one exercise. If you are a fan of unorthodox SAS programming, curious to learn about the variety and flexibility of the SAS language, or an innovative SAS teacher - this article is for you!

Table of contents

*The text was originally published in WUSS 2024 conference proceedings (see [**Jablonski & McMullen 2024**]). This version is extended with SAS Viya examples, has updated title, and is a "republish".

INTRODUCTION

Table look-up has been discussed in multiple articles across many years and at various SAS global, regional, and local conferences and meet-ups. Though the first "officially" titled table look-up related SUGI article by Don Henderson dates back to 1982, see [**Henderson 1982**], the process was discussed already at SUGI.ONE conference in [**Mays 1976**] article. And a few recent ones we can cite are [**Carpenter 2001**], [**Carpenter 2014**], [**Yang et al. 2014**], or [**Iyengar & Horstman 2020**]. The drill is well-known and we are not going to play in presenting any benchmarks or performance tests for different table look-up methods. We rather focus ourselves on the perspective of a teacher, or more precisely, the task of teaching SAS syntax. By pushing fifty shades of grey matter in our brains to the limits, in the article we are going to present as many SAS statements, procedures, functions, and programming techniques as we can to solve this single programming task. While it is well-known that SAS is a flexible language, and creative use of the SAS language often provides multiple ways to solve a problem, we were truly surprised at the number of table look-up methods we could develop. We hope you will enjoy reading the article even longer than 365 days...

THE EXERCISE TO BE SOLVED

So, imagine you are a SAS programming teacher and you are developing a brand new SAS programming class. The exercise we are going to solve is a classic table look-up task, i.e., we have two tables:

- BIG containing a key variable (`ID`) and some data variables (`date`, `value`). The table has several thousand rows[1], each value of the key variable can appear multiple times in the data. We assume that the key variable is *numeric*.
- SMALL containing only one variable and a list of a few random ID values from the table BIG.

We want to select only those rows from the BIG table for which values of the key variable are in the SMALL table. Possible next steps, like: to do some summarization, e.g., calculate a sum or an average of the `value` variable, which could provide an additional bunch of SAS code snippets, will not be discussed in detail.

A note about this paper. As you can observe, the article is vast in terms of both content and concepts. The discussed topics span across mote than fifty pages. We did our best to highlight fundamental ideas for the code presented. But our goal is not to teach these methods, it is to briefly illustrate the amazing variety of techniques that could be used for table look-up, and hopefully inspire readers to dive deeper into approaches they are curious about. This is an unusual goal for a user group paper. We do realize that for less experienced readers some of the explanatory comments may look too brief or be too general. That is why, regardless of whether you are an experienced or novice SAS practitioner, we prepared an extensive list of REFERENCES to provide you with all the necessary supportive resources to make your reading time more enjoyable and your "skills-boost" more efficient.

DATA

We will work with the following data:

(1) a text file containing 11 values, generated by the following snippet (for the coloring convention check Appendix A - code coloring guide):

code: small data to play with

```
/* the small data */
data _null_;
  file "%sysfunc(pathname(WORK))/small.txt";
  put "1 2 3 6 13 17 42 101 303 555 9999";
run;
```

[1] The table is not really big. By modern data standards it is tiny, but is big enough to depict the context, so we call it BIG.

(2) a table located in PostgreSQL database[2], generated by the following snippet:

code: BIG data to play with

```
/* the BIG data */
/* you can use a standard SAS library, PostgreSQL is just as example */
libname PUB POSTGRES
  server="***.***.***.***"
  port=****
  user="******"
  password="*******"
  database="*******"
  schema="*******";
libname PUB list;
/*
proc delete data=PUB.BIG; run;
*/
data PUB.BIG;
  call streaminit(42);
  do year = 2020 to 2024;
    do id = 12345 to 1 by -1;
      date = rand("integer", MDY(1,1,year), MDY(12,31,year));
      value = round(rand("uniform", 100, 200), 0.01);
      if ranuni(17) < 0.9 then output;
    end;
  end;
  format date yymmdd10. value dollar10.2;
  drop year;
run;
```

Those two snippets, though short, allow us to teach an overview of a wide range of SAS language concepts, e.g.,

- DATA steps, PROC steps, and open code,
- idea of SAS libraries (`LIBNAME` statement)
- possibility of using data from both external databases and simple text files,
- the difference between imperative programming in DATA steps and declarative procedures,
- conditional (`IF-THEN-ELSE`) and iterative (`DO-LOOP`) statements,
- functions (`RAND()`) and call subroutines (`CALL STREAMINIT()`),
- idea of formats (`FORMAT` statement), and
- awareness of the macro language existence.

Supportive reading for many of the fundamental concepts can be found for example in [**Henderson 1983**], [**Aster & Seidman 1996**], [**Whitlock 1997**], [**Howard 2004**], [**Whitlock 2006**], [**Whitlock 2007**], [**Kahane 2011**], or [**Dorfman 2013**].

BRINGING DATA TO SAS

As the first step we will bring the data into the SAS session:

(1) Get the BIG data to SAS:

code: bring the BIG data into the SAS session

```
/* get the BIG data into SAS */
data WORK.BIG;
  set PUB.BIG;
  format date yymmdd10. value dollar10.2;
run;
```

[2] PostgreSQL was chosen just as an example, Oracle, Teradata, or any other database would be perfectly fine, eventually even an Excel spreadsheet would work. In fact the BIG can be just stored as a SAS data set.

```
/* order the data */
proc sort data=WORK.BIG;
  by ID date value;
run;
/* know your data */
proc print data=WORK.BIG(obs=42);
run;
proc contents data=WORK.BIG varnum;
run;
proc means data=WORK.BIG;
run;

ods graphics / antialiasmax=45000;
proc sgplot data=WORK.BIG;
  scatter x=date y=value / colorresponse=id
  markerattrs=(size=1 symbol=trianglefilled);
run;
```

Here we can discuss the power of the library engine and its simplicity in use when we want to integrate data from various sources, like an external database for instance. In that context the uniqueness of SAS formats can be highlighted (i.e., the fact that they are very "SAS thing" and external databases cannot honor them). Next we can present some basic SAS procedures and their use in the exploratory (aka, know your data) analysis.

This little piece of code presents the fundamental concept of the DATA step and how the data from one data set can be transformed and moved to the other data set. It also shows that sorting is very easy to perform in SAS with a little help of our friend, the SORT procedure. Basic procedures which allow you to preview data, like the PRINT procedure, or the one to see the metadata and the structure of the data set we are using in the process are also shown, the CONTENTS procedure. The MEANS procedure gives us descriptive statistics and a general overview of numeric variables. Finally the SGPLOT procedure allow us to create a simple graph depicting our data. It is always a good idea to plot your data!

(2) Get the small data to SAS:

code: bring the small data into the SAS session

```
/* get the small DATA into SAS */
data WORK.small_wide;
  infile "%sysfunc(pathname(WORK))/small.txt";
  input ids1 - ids11;
run;
proc print data=WORK.small_wide;
run;
/*
proc transpose
  data=WORK.small_wide
  out=WORK.small_from_wide;
var ids:;
run;
proc print data=WORK.small_from_wide;
run;
*/
data WORK.small;
  infile "%sysfunc(pathname(WORK))/small.txt";
  input ids @@;
run;
proc print data=WORK.small;
run;
```

This little snippet gives us a chance to introduce the INFILE and INPUT statements. We also demonstrate the DATA step's flexibility in reading in text data. The first DATA step produces a data set with one observation and eleven variables (that is why it is called "wide"). The second DATA step produces a data set with one variable and eleven observations. And, commented out, is a simple example of data transposing (more about it and the TRANSPOSE procedure will be shown in later examples).

63 WAYS FOR TABLE LOOK-UP

In this section we are starting the discussion about 63 ways to solve a table look-up task. The list is *more or less* ordered from "easier" to "harder" programs, but not always! Some of those examples are very practical and production ready, others are purely academic, and sometimes even a "scratch your left ear with your right... foot", but we stick to our fundamental idea which is to present possibly the broadest SAS syntax preview as we can!

NAIVE APPROACH

Let us start with something simple. Since we have only 11 values to look-up in our data the naive approach is not a bad idea for the beginning. Naive does not mean it has no educational value!

code: naive approach

```
/* look-up 0, PAINFULLY naive approach */
data WORK.RESULT0;
  set WORK.BIG;

  IF id = 1
  or id = 2
  or id = 3
  or id = 6
  or id = 13
  or id = 17
  or id = 42
  or id = 101
  or id = 303
  or id = 555
  or id = 9999
  THEN OUTPUT;
run;
proc print data=WORK.RESULT0;
run;

/* look-up 1, naive selection */
data WORK.RESULT1;
  set WORK.BIG;

  IF id in (1 2 3 6 13 17 42 101 303 555 9999) THEN OUTPUT;
  /* if id in (...); <- IF-subsetting */
run;

/* look-up 2, naive selection, a bit faster */
data WORK.RESULT2;
  set WORK.BIG;
  WHERE id in (1 2 3 6 13 17 42 101 303 555 9999);

  /* set WORK.BIG(WHERE=( id in (...) )); <- data set option */
run;
```

Those three snippets allow us to discuss conditional processing with IF-THEN-ELSE logic, construction of SAS expressions, ideas of IF-subsetting and WHERE clauses (also to show a comparison of their behavior), as well as the difference between the WHERE statement and the WHERE= data set option. Of course this is

also a good place to talk about the DATA and SET statements, their behavior inside the implicit DATA step loop (aka. the main loop), and the implicit OUTPUT statement at the bottom of a DATA step. This is also a good place to talk about the DATA step compilation phase and execution phase (in the context of IF-subsetting vs. WHERE timing). And of course the Program Data Vector (PDV), illustrated with help of the SAS DATA step debugger, for DMS see [**Riba 2000**] and [**Lavery 2011**], for Enterprise Guide see [**Bayliss & Flynn 2017**] and [**Kim 2018**].

SQL APPROACH

Many new SAS programmers have had some previous exposure to the SQL language, it is rather obvious idea to use this experience as a linker between the two worlds. SQL-heads may be reassured to realize that SAS does SQL.

code: obvious approach

```
/* look-up 3, obvious way, SQL 1 - sub-query */
proc SQL feedback;
  create table WORK.RESULT3 as
  select B.*
  from WORK.BIG as B
  where B.ID in (select s.IDS from WORK.small as s);
quit;
/* look-up 4, obvious way, SQL 2 - Cartesian product */
proc SQL _method;
  create table WORK.RESULT4 as
  select B.*
  from WORK.BIG as B
      ,WORK.small as s
  where B.ID = s.IDS;
quit;
/* look-up 5, obvious way, SQL 3 - JOIN */
proc SQL _tree;
  create table WORK.RESULT5 as
  select B.*
  from WORK.BIG as B
       JOIN /* NATURAL JOIN */
       WORK.small as s
  ON B.ID = s.IDS; /* <no clause> */
quit;
```

When introducing PROC SQL we can explain how combining data can be done using sub-queries, Cartesian product of data sets, and of course joins. But also we can mention about some "SASsy deviations" like the NATURAL JOIN or more or less official options, e.g., FEEDBACK. This snippet on its own could expand into an article or a day long training. Fortunately there are plenty of introductory and advanced papers about the SQL procedure, for example [**Lafler 1992**] or [**Lafler 2017**].

MERGING DATA

Use of the MERGE statement is a classic SAS way of doing table look-ups.

code: SAS obvious approach

```
/* look-up 6, the SAS way, MERGE */
data WORK.RESULT6;
  MERGE
    WORK.BIG(in=B)
    WORK.small(in=S RENAME=(IDS=ID));
  BY ID;
  if S and B;
run;
```

The MERGE statement was discussed in the SUGI.ONE conference article [**Mays 1976**], but it was available since SAS72, watch [**Barr 2018(video)**], so it is even older then the SAS Institute itself (see Appendix E - SAS releases history). The snippet is a good starting place (not the best though) for introductory discussion about the FIRST. and LAST. variables, BY-group processing, and of course the very useful IN= data set option. And of course we have to remember to mention that sorting data over BY variable(s) is necessary. Papers describing how MERGE really works are [**Virgile 1999**] and [**Kahane(2) 2011**].

POINTING OBSERVATIONS

The next snippet introduces non-sequential data reading.

code: pointing observations

```
/* look-up 7, pointing observations, POINT= */
data WORK.RESULT7;
  SET WORK.BIG;

  do POINT = NOBS to 1 by -1;
    set WORK.small POINT=POINT NOBS=NOBS;
    if ID=IDS then
      do;
        OUTPUT WORK.RESULT7;
        GOTO exit;
    end;
  end;
  exit: /* label */
  drop IDS;
run;
```

The program introduces several useful statements and options. First of them all is the OUTPUT statement with the name of a data set to which data are written. The NOBS= compile time option provides information about the number of observations in the read-in data set. The POINT= option allows us to read data sets in non sequential/random order (reversed in this particular case, and without re-sorting!) The "infamous" GOTO statement, see [**Dijkstra 1968**], mentioned rather as a fun fact, because the in depth discussion would require a dedicated article, or a part of one, e.g., [**Luo 2001**]. And the final element, this is the first example of a DATA step that reads data from two data sets with use of two separate SET statements. For many programmers, the realization that a DATA step can have multiple SET statements opens the door to creative programming. The efficiency of the program is rather far from optimal and even jumping out (with the GOTO) from the DO–LOOP as soon as possible does not help here a lot. Some interesting points about the POINT= option can be found in [**Shi & Zhang 1999**].

ARRAYS, VARIABLE LISTS, AND SAS FUNCTIONS

The next few programs try to make the naive approach a bit more robust with the help of arrays, SAS variables lists, and various SAS functions.

code: arrays and variables lists

```
/* look-up 8, conditional SET and ARRAY/VARIABLES LIST */
data WORK.RESULT8_A;
  IF 1 = _N_ then SET WORK.small_wide;
  ARRAY IDS[*] IDS:;
  /* ARRAY IDS[*] IDS1 - IDS11; */ /* or IDS1-numeric-IDS11 */
  DROP IDS:;

  SET WORK.BIG;
  if ID in IDS;
run;
```

```
/* WHICHN and VARIABLES LIST */
data WORK.RESULT8_B;
  IF 1 = _N_ then SET WORK.small_wide;
  DROP IDS:;

  SET WORK.BIG;
  if WHICHN(ID, of IDS1-IDS11);
run;
/* FINDW and CATX */
data WORK.RESULT8_C;
  IF 1 = _N_ then SET WORK.small_wide;
  DROP IDS:;

  SET WORK.BIG;
  if FINDW(catx("|", of IDS:), cats(ID), "|");
run;
```

In each case a wide version of the small data set is conditionally read in the first iteration of the main-loop (1=_N_). In the first one, all IDS1 to IDS11 variables are grouped under one label (the SAS array) and then the array is used with the IN operator. The concept of the SAS array, both the regular one and the temporary array, is fundamental to a SAS programmer becoming a professional. A quick view at www.lexjansen.com gives tons of articles about arrays, to name a few like [**Virgile 1998**], [**Woolridge & Lau 1998**], [**Keelan 2002**], [**Suhr 2005**], [**First & Schudrowitz 2005**], [**Pillay 2015**] or [**Kuligowski & Mendez 2016**]. The second uses a numbered range variable list and the WHICHN() function to do the look-up, see [**Watson & Hadden 2021**]. The third uses a name prefix list inside CATX(), see [**Horstman(2) 2019**], to create a pipe separated string and then search the string with the FINDW() function, see [**Horstman 2015**].

PLAYING WITH TEXT FILES

Not only a data set can be read conditionally, text files can be too.

code: taking data directly from text file

```
filename f8D "%sysfunc(pathname(WORK))/small.txt";
data WORK.RESULT8_D;
  infile f8D;
  if 1 = _N_ then input @@;

  set WORK.BIG;
  if findw(_INFILE_, cats(ID), " ") then output;
run;
filename f8D clear;

filename f8E DUMMY;
data WORK.RESULT8_E;
  FILE f8E;

  if 1 = _N_ then
    do;
      set WORK.small end=EOF;
      PUT IDS1-IDS11 @@;
      drop IDS:;
      /*PUTLOG _FILE_;*/ /* won't work here */
    end;
  set WORK.BIG;
  if findw(_FILE_, cats(ID), " ") then output;
run;
filename f8E clear;
```

The first program reads in the list of values directly from a text file and uses the `_INFILE_` internal variable. The second conditionally puts data into the `_FILE_` internal variable, also the `DUMMY FILENAME` can be introduced. Details available in [**Zdeb 2016**].

PEEKING THE SOLUTION

The last example in this sub-series shows how to directly peek at data from memory.

code: taking data directly from memory

```
data WORK.RESULT8_F;
  IF 1 = _N_ then SET WORK.small_wide;
  DROP IDS:;

  SET WORK.BIG;
  if index (PEEKCLONG(ADDRLONG(IDS1), 88), put(ID, rb8.));

  IF 1 = _N_ then
    do;
    /*---------------------------------*/
      array X IDS:;
      do over X;
        A = ADDRLONG(X);
        Y = put (X, rb8.);
        put X=6. @12 Y= binary. / @12 a= $hex16. @32 a=;
      end;
      drop A Y;
    /*---------------------------------*/
    end;
run;
```

The only new thing here is the fact that SAS allows us to extract the content of the memory directly with help of `ADDRLONG()` and `PEEKLONG()` functions. The conditional snippet at the end of the program was added just to present how memory addressing looks. See [**Dorfman 2009**] for details.

TRYING DOW-LOOP

The next example introduces several ideas, but the DoW-loop is the central one.

code: the DoW-loop

```
/* look-up 9, double SET and DOW-loop, and ARRAY */
data _null_;
  put NOBS=;
  call symputX("NOBS9", nobs, "G");
  stop;
  set WORK.small NOBS=NOBS;
run;

data WORK.RESULT9;
  ARRAY _IDS_[&NOBS9.] _TEMPORARY_; /* &NOBS9. = 11 <- from NOBS */
  do until(EOF_S);
    SET WORK.small end=EOF_S curobs=curobs;
    _IDS_[curobs] = IDS;
    drop IDS;
  end;

  do until(EOF_B);
    SET WORK.BIG end=EOF_B;
    if ID in _IDS_ then output;
  end;
stop;
run;
```

The first DATA step shows how macro variables can be generated from data using the `CALL SYMPUTX()` subroutine. The second introduces the concept of a `_TEMPORARY_` array that, in contrary to a "regular" SAS array being just a PDV variables quick "referencer", is a dedicated memory block allocated during the DATA step compilation phase. Two `DO-UNTIL` loops are examples of the so called DoW-loop, a technique of conditional sequential data reading popularized by Ian Whitlock and exceptionally well described in [**Dorfman & Vyverman 2009**], but known since at least [**Henderson 1983**]. The first `DO-UNTIL` loop populates the temporary array using data from the small data set, and the second one iterates over BIG data set. The `STOP` statement is an explicit termination of the DATA step. Discussion of macro variables can be found in [**Zender 2013**] or [**Lavery 2007**]. And the ultimate source of macro language knowledge is the [**Carpenter 2016**].

MULTIPLE DATA SETS

Up to now we have shown DATA steps having multiple `SET` statements, but each reading one data set. When multiple data sets are provided to the `SET` statement it reads data sets sequentially, one after another. This functionality combined with the `IN=` data set option and temporary arrays gives us a very interesting program (again `CALL SYMPUTX()` is used).

code: multiple data sets read at once

```
/* look-up 10, SET and multiple data sets */
data _null_;
  put NOBS=;
  call symputX("NOBS10", nobs, "G");
  stop;
  set WORK.small NOBS=NOBS;
run;

data WORK.RESULT10;
  ARRAY _IDS_[&NOBS10.] _TEMPORARY_;

  SET WORK.small(in=S) WORK.BIG curobs=curobs;
  if S then _IDS_[curobs] = IDS;
  else
    if ID in _IDS_ then output;

  drop ids;
run;
```

INTERLEAVING DATA SETS

When the `SET` statement reading multiple data sets is combined with the `BY` statement the process of reading data is alternated. Instead of reading data from the two data sets sequentially like in the previous example, the records from both data sets are read in order defined by values of the `BY` statement variables. These two programs present the concept of data set interleaving.

code: interleaving

```
/* look-up 10, cont., interleaving data in SET statement */
data WORK.RESULT10_A;
  SET
    WORK.small(in=S RENAME=(IDS=ID))
    WORK.BIG(in=B);
  BY ID;

  if S and FIRST.ID then _check+ID;
  if B and _check=ID then output;
  if LAST.ID then _check=.;

  drop _:;
run;
```

```
data WORK.RESULT10_B;
  DO UNTIL(last.id);
    set
      WORK.small(in=S RENAME=(IDS=ID))
      WORK.BIG(in=B);
    by ID;

    if FIRST.ID then _check=S;
    if B and _check then output;
  END;

  drop _:;
run ;
```

The interleaving process reads data from multiple data sets in order dictated by variables in the `BY` statement. The first DATA step uses the `FIRST.` and `LAST.` logic to select the observations to be output. The second one additionally executes the process in a DoW-loop which allows us to avoid explicitly setting the value of the `_check` variable to missing.

SOME MORE DOW-LOOPING

The subsequent snippets provide more examples of how the DoW-loop can be used to perform a table look-up.

code: the DoW-loop, continued

```
/* look-up 11, DoW-loop, cont. */
/* (extra assumption that all IDs exist in BIG) */
data WORK.RESULT11_A;
  SET WORK.small;
  drop IDS;

  do until(last.ID and ID=IDS);
    SET WORK.BIG;
    by ID;
    if ID = IDS then output;
  end;
run;
/* (NO extra assumption that all IDs exist in BIG) */
/* options mergenoby=nowarn; */
data WORK.RESULT11_B;
  SET WORK.small;
  drop IDS nextID;

  do until(nextID>IDS) ;
    MERGE
      WORK.BIG
      WORK.BIG(FIRSTOBS=2 keep=ID rename=(ID=nextID));
    if ID=IDs then output;
  end;
run;
/* options mergenoby=error; */
```

DATA step number one uses the SMALL data set as a driving file which determines DoW-loop iterations over the BIG data set read by the `SET` statement. An additional assumption here is that all values from the SMALL data set have to be in the BIG one. DATA step number two does not need that additional assumption since it utilizes the "future reading" of observations with the `BY`-less `MERGE` statement and the `FIRSTOBS=` data set option. Since in some industries the `BY`-less `MERGE` statement is a bit infamous the `MERGENOBY=` option can help. The approach of using the `MERGE` statement with the `FIRSTOBS=` data set option is discussed in [**Keintz 2017**].

USER-DEFINED FORMATS

User-defined formats have tons of use cases, table look-up task being one among them.

code: formats

```
/* look-up 12, User-Defined Format */
proc format;
  VALUE myFormat
    1, 2, 3, 6, 13, 17, 42, 101, 303, 555, 9999 = "Y"
    OTHER = "N"
    ;
run;

data WORK.RESULT12;
  SET WORK.BIG;
  where put(ID,myFormat.) = "Y";
run;

/* look-up 13, User-Defined Format from data */
data input_control_SAS_data_set;
  set WORK.small END=EOF;
  rename IDS=START;
  FMTNAME="myFormatFromData";
  LABEL="Y";
  TYPF="F";
  output;
  IF EOF;

  LABEL="N";
  HLO="O";
  output;
run;

proc format CNTLIN=input_control_SAS_data_set;
run;

data WORK.RESULT13;
  SET WORK.BIG;
  where put(ID,myFormatFromData.) = "Y";
run;
/*
proc format LIB=WORK;
  select myFormat myFormatFromData;
run;
*/
```

The first program uses an explicitly defined format where the programmer provides a list of values and labels. The second one uses a data-driven approach where a specific input control data set is prepared and then used by the `FORMAT` procedure. Use cases for the `FORMAT` procedure are so popular that there are tons of articles dedicated to it, to name a few [**Patton 1998**], [**Shoemaker 2001**], [**Shoemaker 2002**], [**Eason 2005**], [**Wright 2007**], or [**Bilenas 2008**].

DIRECT ADDRESSING

Since the ID variable is numeric we can easily use a technique called direct addressing.

code: direct addressing - array

```
/* look-up 14 A, ARRAY and direct addressing */
options symbolgen;
data _null_;
  set WORK.small END=EOF;
  retain min max;
  max = max(max, IDS);
  min = min(min, IDS);
  if EOF;
  put max= min=;
  call symputX("max14",max,"G");
  call symputX("min14",min,"G");
run;


data WORK.RESULT14_A;
  ARRAY T[&min14.:&max14.] _temporary_;
  do until(EOF_S);
    set WORK.small end=EOF_S;
    T[IDS] = 1;
  end;

  do until(EOF_B);
    SET WORK.BIG end=EOF_B;
    if &min14. <= ID <= &max14. then
      if T[ID] then output; /* this is direct addressing */
  end;
stop;
drop IDS;
run;
```

The technique uses the fact that values of the ID variable can be used to directly point to array cells. The cells contain binary information (1 or null) indicating whether the given ID exists in the small data set. The &min. and &max. macro variables are used to optimize the array's size. To calculate minimum and maximum we are using MIN() and MAX() functions and SAS distinguishes them from min and max variables (a side note, for variables defined in DATA step code we use the RETAIN statement to prevent their values from being set to missing at the beginning of each iteration of the implicit DATA step loop). And the SYMBOLGEN option allows us to preview the values of macro variables used in the program. The idea of direct addressing, a predecessor of the idea of hash tables, was described by Paul Dorfman in [**Dorfman 2001**] article.

We can also go one step further and instead of using an array, we can use a *bitmap* for direct addressing. Instead of using all eight bytes of an array cell for storing one value of 1, we can use 32 (or, after some modifications, even 53[3]) *bits* from each cell of an array to mark a value. This approach gives us significant memory savings (32 or 53 times).

code: direct addressing - bitmap

```
/* look-up 14 B, BITMAP and direct addressing */
%let M = 32; /* bitmap size (32 bits) */
%let KL = 1; /* ID (key) low value */
%let KH = 12345; /* ID (key) high value */
%let R = %eval (&KH - &KL + 1); /* keys range */
%let D = %sysfunc (ceil (&R / &M)); /* dim of bitmap array*/
%put &=M. &=KL. &=KH. &=R. &=D.;

```

[3] Hmm... 53... coincidence? See [**Jablonski & McMullen 2024**].

```
data WORK.RESULT14_B ;
  /* Initialization of bitmap and bitmask */
  array BM [&D] _temporary_ (&D.*0);
  array bitmask [0:&M] _temporary_;
  do B = 0 to &M;
    bitmask[B] = 2**B;
  end;
  /* Mapping IDs to Bitmap */
  do until(EOF_S);
    set WORK.small end=EOF_S;
    C = int (divide (IDS - 1, &M)) + 1; /* find bitmap cell */
    B = 1 + mod (IDS - 1, &M); /* find bit in cell */
    BM[C] = BOR (BM[C], bitmask[B - 1]); /* activate bit */
    N_mapped + 1;
  end;
  /* Searching IDs in Bitmap */
  do until(EOF_B);
    SET WORK.BIG end=EOF_B;
      C = int (divide (ID - 1, &M)) + 1;
      B = 1 + mod (ID - 1, &M);
      ActiveBit = BAND (BM[C], bitmask[B - 1]) ne 0;
      N_found + ActiveBit;
      if ActiveBit then output;
  end;
  put N_Mapped= N_found=; /* Number of mapped and found values */
stop;
keep ID date value;
run;
```

Macro variables were used to increase code maintainability. Bitmap size is calculated based on `ID` values range in the BIG data set. As can be seen, bitmaps are *not* internal SAS data structure, but with a few additional lines of code, they can be implemented in DATA step based on arrays. The snippet presented is just one example of all palette of bitmaps, in-depth and precise explanation of the technique can be found in [**Dorfman & Shajenko 2019**]. Though it seems very advanced, in fact the principles are much easier to grasp than it may seem. It is a very powerful technique worth learning.

HASH TABLES

Hash tables are execution phase dynamic objects which have proven their extraordinary usefulness in SAS programs, one common use case being a table look-up.

code: hash tables

```
/* look-up 15, HASH TABLES */
data WORK.RESULT15;

  if 0 then set WORK.small;
  DECLARE HASH H(dataset:"WORK.small");
  H.defineKey("IDS");
  /* H.defineData("IDS"); */
  H.defineDone();

  do until(EOF_B);
    SET WORK.BIG end=EOF_B;

    if 0=H.CHECK(key:ID) then output;
  end;
stop;
drop IDS;
run;
```

The DECLARE statement is used to initiate the HASH object H. The hash object H is built from the SMALL data set with variable IDS as a key of the object. The .CHECK() method tests if a given value of the ID variable exists in H. A successful check returns zero as a value. There are many articles and a few books discussing hash objects and their properties starting with [**Dorfman & Snell 2002**] and [**Dorfman & Snell 2003**], [**Secosky & Bloom 2007**], [**Loren & DeVenezia 2011**], [**Dorfman 2014**], [**Dorfman & Henderson 2015**], and ending with [**Carpenter 2012**] and [**Dorfman & Henderson 2018**] being among the most valuable. Just to highlight it one more time, table look-up is just a surface scratching of hash table functionality!

SMART-NAIVE WITH MACRO VARIABLES

We have seen a few ways to automate the naive approach, here are a few more.

code: macro variable lists and arrays

```
/* look-up 16, naive selection - but smart, v1 */
/* macro variable with values list */
proc SQL noprint;
  select distinct IDS
  into :IDS16_A separated by " "
  from WORK.small;
quit;

data WORK.RESULT16_A;
  set WORK.BIG;
  WHERE id in (&IDS16_A.);
run;

/* macro variable array */
proc SQL noprint;
  select distinct IDS
  into :IDS16_B1-
  from WORK.small
  ;
  %let N16B=&sqlobs.;
quit;

%macro loop(n);
  %do n = 1 %to &n.;
    &&IDS16_B&n.
  %end;
%mend;

data WORK.RESULT16_B;
  set WORK.BIG;
  WHERE id in (
      %loop(&N16B.)
    );
run;
```

A macro variable with a list of values can be created, among many other ways, with help of the SQL procedure and the INTO clause. The separated by part informs SAS that values read in from the data set should be separated by a space character. Version "B" also uses PROC SQL but, instead of creating one macro variable with a space separated list of values, it creates a list of macro variables with a common prefix and numeric suffix. Such lists are commonly called macro variable arrays. We use indirect referencing in the %loop() macro to iterate over the macro variable array. A very good introduction to the subject of macro variable arrays can be found in [**Fehd 2003**], [**Horstman 2019**], [**Renauldo 2018**], [**Carpenter 2017**], or [**Jablonski 2024**].

CALLING THE EXECUTIONER

While the macro language is a common way to generate SAS code, SAS also provides other ways to generate code. First of them uses the `CALL EXECUTE()` subroutine.

code: call execute

```
/* look-up 17, naive selection - but smart, v2 */
data _null_;
  call execute("
  data WORK.RESULT17;
    set WORK.BIG;
    WHERE id in (
  ");

  do until(EOF);
    set WORK.small end=EOF;
    call execute(IDS);
  end;

  call execute("); run;");
stop;
run;
```

DATA step code is fed in the form of a text string to the `CALL EXECUTE()` subroutine, including the list of values from the SMALL data set provided by a DoW-loop. The `CALL EXECUTE()` subroutine is a well-known tool in every SAS programmer's toolbox. Its usage was discussed in [**Whitlock(2) 1997**], [**Virgile 1997**], and [**Michel 2005**].

INCLUDING CODE

The next one is the `%INCLUDE` statement which despite the fact it contains the percent sign, is *not* a macro language statement.

code: %include

```
/* look-up 18, naive selection - but smart, v3 */
filename F18 TEMP;
data _null_;
  file F18 ;
  put "WHERE id in (";

  do until(EOF);
    set WORK.small end=EOF;
    put IDS;
  end;

  put ");";
stop;
run;

data WORK.RESULT18;
  set WORK.BIG;
  %include F18 / source2;
run;
filename F18 CLEAR;
```

The first DATA step generates a text file which contains a valid SAS statement (this is a requirement for the `%INCLUDE` to work) and in the second step the `%INCLUDE` statement is used to include the file's content into the program. The `SOURCE2` option instructs SAS to print out the included file content in the log. Information about the `%INCLUDE` statement ca be found in [**Carpenter 2002**].

SUBMITTING A LINE

The last but not least code generation technique to be shown, DOSUBL()[4] function, is a younger sibling of the CALL EXECUTE() subroutine.

code: dosubl

```
/* look-up 19, naive selection - but smart, v4 */
data _null_;
  length text $ 32767;
  text = "data WORK.RESULT19; set WORK.BIG; WHERE id in (";

  do until(EOF);
    set WORK.small end=EOF;
    text =  catx(" ", text, IDS);
  end;

  text =  catx(" ", text, "); run;");
  put text;
  rc = DOSUBL(text);
stop;
run;
proc print data = WORK.RESULT19;
run;
```

Similar to CALL EXECUTE(), DATA step code is constructed in a text variable and then passed as an argument to the DOSUBL() function. The differences are: 1) in CALL EXECUTE() code text can be provided "in parts" to multiple calls vs. in DOSUBL() it has to be one snippet, 2) CALL EXECUTE() writes provided code to an input stack and the code is executed after the DATA step ends vs. DOSUBL() creates a "magical side SAS session" and executes the provided code on the fly, during the execution of the calling DATA step. In depth discussion about the DOSUBL() function can be found in [**Langston 2013**] and [**McMullen 2020**].

USER-DEFINED FUNCTIONS

If you think that formats, arrays or hash tables have tons of use cases you have not seen the power of User-Defined Functions.

code: udf

```
/* look-up 20, User-Defined Functions, v1 */
proc FCMP outlib=WORK.F20.P;
  function F20(x);
    array A[1] / NOSYMBOLS;
    static A read;
    if NOT read then do;
        rc = READ_ARRAY("WORK.small", A, 'IDS');
        read = 1;
      end;
    /* small is sorted, so binary search is possible */
    l=1; h=dim(A);
    do while(l<=h);
      i = int((l+h)/2);
      if (A[i] = x) then return(1);
                    else if (A[i] < x) then l=i+1;
                                       else h=i-1;
    end;
    return(0);
  endfunc;
run;
```

[4]The name DoSubL comes from the "DO SUBmit Line" phrase.

```
options append=(cmplib=WORK.F20);
data WORK.RESULT20;
  set WORK.BIG;
  WHERE 1 = F20(id);
run;
/* look-up 21, User-Defined Functions, v2 */
proc FCMP outlib=WORK.F21.P;
  function F21(IDS);
    DECLARE HASH H(dataset: "WORK.small");
      rc = H.defineKey("IDS");
      rc = H.defineDone();
    return(NOT H.CHECK());
  endfunc;
run;

options append=(cmplib=WORK.F21);
data WORK.RESULT21;
  set WORK.BIG;
  WHERE 1 = F21(id);
run;
```

The FCMP procedure (Function CoMPiler) allows us to write functions and subroutines which can be used in DATA steps or even SQL queries. The FCMP procedure provides dedicated tools which allow us to read data from data sets, for example the READ_ARRAY() function, and hash tables too. Functions defined with those tools behave like dictionaries and can be used in filtering conditions. The NOSYMBOLS option instructs SAS to treat the array analogously to a DATA step _TEMPORARY_ array. The STATIC keyword, keeping a variable's value across subsequent calls to the same function, somehow alike to DATA step's RETAIN maintaining value across DATA step implicit loop (this can be a good occasion to discuss and compare them). To learn more about user-defined functions you can reach out to [**Secosky 2007**], [**Eberhardt 2009**], [**Secosky 2012**], [**Rhoads 2012**], [**Henrick et al. 2013**], [**McNeill et al. 2018**], [**Carpenter 2018**], and [**Hughes 2024**] book.

KEYS TO SOLUTION

Since the advent of hash tables or even earlier, PROC SQL, the next technique might be less popular but it is still helpful to teach students about indexing in SAS.

code: key=

```
/* look-up 22, INDEXED SET + KEY= + KEYRESET= */
proc datasets lib=WORK noprint;
  modify BIG;
    INDEX CREATE ID; /* index delete ID; */
  run;
quit;

data WORK.RESULT22;
  set WORK.small(rename=(IDS=ID));
  reset = 1;
  do while (NOT _iorc_);
    set WORK.BIG key=ID keyreset=reset;
    if NOT _iorc_ then output;
  end;
  _error_ = 0; _iorc_ = 0;
run;
```

The program uses the DATASETS procedure to create an index on the ID variable to improve the speed of indirect data access done with the KEY= option. The SMALL data set provides values which are searched in the BIG data set when KEY= is used, since there can be multiple observations with one value of the ID

variable the process is looped over until the _IORC_ variable is different than zero. This type of table look-up is discussed in [**Aker 2000**]. More about the DATASETS procedure can be found in [**Raithel 2018**] and SAS indexes are best described in [**Clifford 2005**].

MODIFYING RESULT

The same level of popularity decrease for table look-up task can be observed with the MODIFY statement, but beware and do not underestimate the power of in place modifications.

code: modify

```
/* look-up 23 A, MODIFY */
data WORK.RESULT23_A;
  set WORK.BIG(obs=0) WORK.small(rename=(IDS=ID));
run;

proc datasets lib=WORK noprint;
  modify RESULT23_A;
    index create ID;
  run;
quit;

data WORK.RESULT23_A;
  MODIFY WORK.RESULT23_A WORK.BIG;
  by ID;

  if _iorc_=0 then output;
  _error_ = 0; _iorc_ = 0;
run;
proc print data=WORK.RESULT23_A(where=(value)); /* ! */
run;
```

Additional pre-processing is needed to concoct the driving data set with data out of SMALL and the structure taken from BIG. The concocted data set has the list of required IDS values but both date and value are missing. Whenever an ID value already existing in the result is encountered in the BIG data set a new observation is added to the result. So we have to remember to ignore observations with missing dates and values in the result data set (the WHERE= data set option with filtering condition). Read more about the MODIFY statement in [**Mack 2008**], and for some unorthodox use cases see [**Dorfman 2018**] and [**Bremser 2022**].

INTEGRITY OF IT ALL

Since we already mentioned the DATASETS procedure and SAS data sets' indexes we cannot forget about their siblings, integrity constraints. The next two snippets use different types of integrity constraints (IC for short). This first uses the CHECK type IC which, as the name suggests, checks data with provided where condition. The condition data are provided in form of already mentioned macro variable list.

code: IC of type check

```
/* look-up 23 B, Integrity Constraints - CHECK with where condition */
proc SQL noprint;
  select distinct IDS
  into :IDS23_B separated by " "
  from WORK.small;
quit;

data WORK.RESULT23_B;
  stop;
  set WORK.BIG;
run;
```

```
proc datasets lib=WORK noprint;
  modify RESULT23_B;
    IC create IC_of_type_check = check(where=(ID in (&IDS23_B.)));
  run;
quit;
proc append base=WORK.RESULT23_B
            data=WORK.BIG;
run;
```

The second focuses on using the small data set as a "gate keeper" dictionary which allows us to append data only if values are in the small data set.

code: IC of type foreign key

```
/* look-up 23 C, Integrity Constraints - FOREIGN KEY */
proc SQL noprint;
  create table work.IC_PK_small as
  select distinct IDS
  from WORK.small;
  create unique index IDS on work.IC_PK_small(IDS);
  alter table work.IC_PK_small add primary key (IDS);
quit;
data WORK.RESULT23_C;
  stop; set WORK.BIG;
run;
proc datasets lib=WORK noprint;
  modify RESULT23_C;
    IC create IC_of_type_foreignkey =
      FOREIGN KEY (ID) REFERENCES work.IC_PK_small ON UPDATE RESTRICT;
  run;
quit;
proc append base=WORK.RESULT23_C
            data=WORK.BIG;
run;
```

A common part for both snippets is the fact that when we are trying to add improper data to the result data set SAS will give us a `WARNING:` message that our data violated rules. Also the `MSGLEVEL=i` option provides interesting insights on the `APPEND` procedure behavior.

The data validation process can be greatly improved and simplified with use of ICs, example of such solutions can be found in [**Raithel(2) 2018**] Integrity constraints can be created both by `PROC DATASETS` and `PROC SQL`, see [**Franklin & Jensen 2000**], or [**Fickbohm 2006**] and [**Fickbohm 2007**].

FETCHING THE ANSWER

If you have not had the chance to work with SAS SCL (Screen Control Language) input/output functions the next example might be something new for you. The trick here is, that SCL's functions can be also used in DATA steps to perform data extraction.

code: I/O functions

```
/* look-up 24, OPEN + FETCH */
data WORK.RESULT24;
  set WORK.BIG(obs=0) WORK.small;
  did = OPEN("WORK.BIG(where=(id=" !! put(IDS,best32.) !! "))");
  if did then do;
    CALL SET(did);
    do while (0=FETCH(did)); output; end;
  end;
  did = CLOSE(did);
run;
```

The approach used here is somehow similar to that in interleaving of data sets where the SMALL data set is used as a driving file which dictates which values should be extracted from the BIG one. The difference is that in this case the I/O functions play the main role. More about those functions can be found here in [**Scerbo 1992**], [**Manickam 2012**], or [**Mukherjee 2019**].

YOUNGER SIBLING

Hollywood loves sequels, also SAS Institute seemed to adopt the trend when SAS introduced the DS2 procedure (Data Step Two).

code: proc ds2

```
/* look-up 25, PROC DS2 */
PROC DS2;
  data WORK.RESULT25_A / overwrite=yes;
    method run();
      SET {select B.*
           from WORK.BIG as B
           join
           WORK.small as s
           on B.ID = s.IDS
          };
      output;
    end;
  enddata;
  run;

  data WORK.RESULT25_B / overwrite=yes;
    method run();
      MERGE
        WORK.BIG(in=B)
        WORK.small(in=S rename=(IDS=ID))
      / RETAIN; /* ! */
      BY ID;
      if (B and S ) then output;
    end;
  enddata;
  run;

  data WORK.RESULT25_C/ overwrite=yes;
    declare double IDS rc;
    declare package hash H(8,'WORK.small');
    drop IDS rc;

    method init();
      rc = H.defineKey('IDS');
      /*rc = H.defineData('IDS');*/
      rc = H.defineDone();
    end;
    method run();
      set WORK.BIG;
      if (0 = H.check([ID])) then output;
    end;
  enddata;
  run;
QUIT;
proc print data=WORK.RESULT25_A;
proc print data=WORK.RESULT25_B;
proc print data=WORK.RESULT25_C;
run;
```

The DS2 procedure feels like a "spin-off" to the "classic" DATA step. The majority of concepts overlap with the DATA step, but some things that look similar work differently (e.g. merging), there are some new features introduced (e.g. object-oriented approach and thread processing), but also some "good old ones" missing (interaction with external files). Because of the rather vast overlap with the DATA step we present only three snippets. They highlight interesting or surprising features like 1) inline SQL queries, 2) a difference in the MERGE statement, and 3) the concept of the DS2 procedure specific packages. An interesting discussion comparing the DATA step and the DS2 procedure can be found in [**Hughes 2019**]. The ultimate handbook of PROC DS2 is [**Jordan 2018**].

YOUNGER SIBLING'S FRIEND

Like Joey Tribbiani to Chandler Bing, in the context of external databases connectivity the DS2 procedure seems to be entangled with PROC FEDSQL. The FEDSQL procedure provides the SAS implementation of the ANSI SQL:1999 core standard (while PROC SQL follows most of the guidelines set by the ANSI:1992).

code: fedsql

```
/* look-up 26, PROC FedSQL; */
PROC FedSQL;
  drop table WORK.RESULT26 FORCE;
  create table WORK.RESULT26 as
    select B.*
    from WORK.BIG as B
    join
    WORK.small as s
    on B.ID = s.IDS;
QUIT;
proc print data=WORK.RESULT26;
run;
```

Here we present only one JOIN example. Discussion of PROC FEDSQL (in various configurations) can be found in [**Mohammed et al. 2015**], [**Huffman et al. 2018**], or [**Morioka 2019**].

THE MATRIX

In the 4GL we are accustomed to using loops (implicit or explicit) to process data, either in data sets (with SET statement) or in arrays, but if you ever wanted to try vectorized programming the IML procedure is the answer. The IML stands for Interactive Matrix Language.

code: iml

```
/* look-up 27, PROC IML; */
PROC IML;
  use WORK.BIG;
    read all var {id date value};
  close WORK.BIG;
  use WORK.small;
    read all var {ids};
  close WORK.small;
  TF = LOC(ELEMENT(ID, IDS));
  id = id[TF];
  date = date[TF];
    mattrib date format=yymmdd10.;
  value = value[TF];
    mattrib value format=dollar10.2;
  create WORK.RESULT27 var {"id" "date" "value"};
    append from id date value;
  close WORK.RESULT27;
QUIT;
proc print data=WORK.RESULT27;
run;
```

Interaction between IML and 4GL works in both directions, so data can be read-in and written-out. The variables are read into vectors, observations we are looking for are located with help of the LOC() and ELEMENT() functions, a logical vector TF (True/False) is created and used to filter out only interesting elements. If you are thinking about learning IML there is only one name to mention - Rick Wicklin, his blog "The DO Loop" (https://blogs.sas.com/content/iml/) is an infinite source of IML knowledge, also the [**Wicklin 2010**] book is an excellent source.

The same way the "Matrix" could alter reality the IML can influence DATA step processing. The second way the IML can be used to execute look-up is use of the SUBMIT-ENDSUBMIT block:

code: iml + submit block

```
/* look-up 27 B, PROC IML; */
PROC IML;
  use WORK.small;
  read all var ids;
  close WORK.small;

  reset log;
  show names;
  print ids;

  submit listOfIDs=IDS / ok=OK;
    data WORK.RESULT27B;
      set WORK.BIG;
      where id in (&listOfIDs.);
    run;
    title "Submit from IML";
    proc print data = WORK.RESULT27B;
    run;
    title;
  endsubmit;
  print OK;
QUIT;
```

TRANSPOSING GIVES FLEXIBILITY

At the beginning of this article we wrote that some of the examples we present are a bit like "scratching left ear with right... foot". You may say it is quite uncomfortable on one side but on the other when you see a man doing such thing you may think: "that's flexibility!" The upcoming three examples may look like such a case. But we want to discuss them just to show how flexible the SAS language is!

The first one uses the TRANSPOSE procedure and the APPEND procedure.

code: transpose

```
/* look-up 28, PROC TRANSPOSE and APPEND; */
proc sort
  data = WORK.BIG
  out  = WORK.BIGSORTED28;
by date id value;
run;

proc transpose
  data = WORK.BIGSORTED28
  out  = WORK.BIGTRANSPOSED28(drop=_:)
  prefix = ID;
   by DATE;
   id ID;
   var value;
run;
```

```
proc transpose
  data = WORK.small
  out  = WORK.smallTRANSPOSED28(drop=_:)
  prefix = ID;
   var ids;
   id ids;
run;
/* shell data set */
data WORK.smallTRANSPOSED28;
  date = .; format date yymmdd10.;
  set WORK.smallTRANSPOSED28;
  stop;
run;
filename D DUMMY; proc printto log=D;run;
PROC APPEND
  BASE = WORK.smallTRANSPOSED28
  DATA = WORK.BIGTRANSPOSED28
  FORCE;
RUN;
proc printto;run; filename D clear;

proc transpose
  data = WORK.smallTRANSPOSED28
  out  = WORK.RESULT28(where=(value1))
  name = ID
  prefix = value;
   by DATE;
   var id:;
run;

proc sort
  data = WORK.RESULT28
  SORTSEQ=LINGUISTIC(NUMERIC_COLLATION=ON);
by id date value1;
run;
/* ! */
data WORK.RESULT28;
  length newID 8;
  set WORK.RESULT28;
  newID = input(compress(ID, , "KD"), best32.);
  drop ID;
  rename newID=ID value1=value;
run;
```

The presented program uses the `TRANSPOSE` and the `APPEND` procedures. First, the BIG data set is re-sorted by the `date` variable. Then the `TRANSPOSE` procedure re-shapes data in groups by the `date` variable values, in such a way that each value of the `ID` variable is used to create a new variable named from concatenation of constant text prefix "ID" and the value of `ID`. Result looks more or less like this:

date	ID1	ID2	ID3	ID4	ID5	ID6	⋯
2024-09-03	.	1	.	2	.	.	
2024-09-04	3	.	.	4	.	.	
2024-09-05	.	.	5	6	.	7	
2024-09-06	.	8	9	.	.	.	
⋮	.	.	.	.	.	.	⋱

with respect to the variables' order. Two subsequent steps, `PROC TRANSPOSE` and DATA step, transform the SMALL data set to a similar form, which in our case is: data set with 0 (zero) observations and exactly those

variables named: date, ID1, ID2, ID3, ID6, ID13, ID17, ID42, ID101, ID303, ID555, ID9999. Next step, the APPEND procedure is the place where the magic happens, "the Jedi knight tricks" to be precise, after all we are using the FORCE. The APPEND procedure, by default, does not allow us to attach new data to the old one if their structures are different, but if the FORCE option is used, for those variables which are common, data are appended. So all observations from the transposed BIG data set are appended to a shell data set created from the SMALL, but only for variables date, ID1, ID2, ID3, ID6, ID13, ID17, ID42, ID101, ID303, ID555, and ID9999. So we are basically cutting off (like with a lightsaber) variables that we do not want in the result. The process of forcing SAS to append incompatible data sets has a price. The price is an avalanche of warning messages in the LOG, thus we wrap-up the APPEND procedure with the "dummy file & PROC PRINTTO" sandwich, which redirects all that ugliness to the void. Eventually we end up with a data set which looks, again more or less, like that:

date	ID1	ID2	ID3	ID6	···	ID555	ID9999
2024-09-03	.	1	.	.	.	.	9999.1
2024-09-04	3	.	.	.	.	555.1	.
2024-09-05	.	.	5	7	.	555.2	9999.2
2024-09-06	.	8	9	.	.	555.3	.
⋮	.	.	.	.	.	.	⋮

which is almost what we want. One more transposition and sorting, and we will almost get what we need. The famous Rolling Stones song goes: *You can't always get what you want; But if you try sometimes, well, you might find; You get what you need;*

In our case, with one extra little DATA step we can get what we want. An additional note has to be added! Though this process is pretty automatic, in a sense that we do not have to provide a number of metadata information, it is not very I/O efficient. The transposition step "explodes" our BIG(1.4MB) data set to 176.3MB, more than hundred and twenty-five times! But, as we mentioned earlier, this one is not about production ready code, it is about SAS "flexibility".

PIVOTING POINT OF VIEW

The next one uses the TABULATE and the MEANS procedures with the CLASSDATA= option.

code: classdata=

```
/* look-up 29, PROC TABULATE and MEANS - CLASSDATA= */
proc sort
  data=WORK.BIG
  out=WORK.SORTED29;
by date id value;
run;
ods select NONE; filename D DUMMY; proc printto log=D;run;
proc tabulate
  CLASSDATA=WORK.small(rename=(IDS=ID)) EXCLUSIVE
  data=WORK.SORTED29
  OUT=WORK.RESULT29_A(drop=_: rename=value_sum=value where=(.z<value));
   class ID;
   by DATE;
   var value;
   table ID,value*sum=" ";
run;
ods select ALL; proc printto;run; filename D clear;
proc print data=WORK.RESULT29_A;
run;
data WORK.BIGview29 / view = WORK.BIGview29;
  set WORK.BIG;
  _obs = _n_;
run;
```

```
proc means
  CLASSDATA=WORK.small(rename=(ids=id)) EXCLUSIVE
  data=WORK.BIGview29
  noprint nway;
   class ID;
   by _obs date;
   var value;
   OUTPUT
    out=WORK.RESULT29_B(drop=_: where=(not missing(value))) sum=;
run;
proc print data=WORK.RESULT29_B;
run;
```

Programs in this snippet use procedures which traditionally are considered to be summary/reporting/statistical ones. The TABULATE procedure is SAS' version of Excel's Pivot table (but better), the MEANS procedure is a summarizing procedure, in its basic use case similar to R's `summary()` function. By the way, in SAS, PROC MEANS has a twin named PROC SUMMARY. In both cases the trick here is done with use of the CLASSDATA= and the EXCLUSIVE options. The first points the small data set values to be class variables to analyze, the second that they are the *only* one. This gives us the filtering. Both procedures provide a result data set with the OUT= option, but, similarly to the TRANSPOSE procedure, the TABULATE procedure has to be wrapped-up inside the "select none, dummy & PROC PRINTTO" sandwich. These examples, in contrary to the previous one, are not resources hungry. Good introductory reading about the TABULATE procedure is [**Winn Jr. 2008**](exceptional references list) or [**Wright 2008**]. And about the MEANS procedure see [**Karp 2004**].

SORT THE PROBLEM OUT

The next one uses the SORT procedure.

code: sorting

```
/* look-up 30, unique values */
/* set a technical macro variable with */
/* expected maximum observations in one ID group */
%let maxDup=7;
data WORK.smallDup / view = WORK.smallDup;
  set WORK.small(rename=(IDS=ID));
  do d = 1 to &maxDup.;
    output;
  end;
run;
data WORK.BIGview30 / view = WORK.BIGview30;
  set WORK.BIG;
  by ID;
  if first.ID then d=0; d+1;
run;
/* a view built on views */
data WORK.BSv30 / view = WORK.BSv30;
  set WORK.smallDup WORK.BIGview30 ;
run ;
proc sort
  NODUPKEY
  data   = WORK.BSv30
  out    = _null_
  DUPOUT = WORK.RESULT30(drop=d);
by id d;
run;
proc print data=WORK.RESULT30;
run;
```

This snippet's "magic" is in the DUPOUT= option. An option which indicates where all observations with duplicated sorting key values should be stored. The first pre-processing DATA step creates a view based on the SMALL data set with each value multiplied &maxDup. times and the d variable stores the sequence of numbers for each IDS variable value. The maxDup macro variable stores a number which is an expected maximum number of observations for a single ID variable group in the BIG data set. The second pre-processing DATA step just adds the new variable d which stores the observation number in each ID variable group. Now we have a new "artificial" primary key built on variables ID and d in both views. The third pre-processing DATA step creates a view which combines data from the first and the second. Finally we do the sorting. The only ID and d pairs which have duplicates are those which exists in both the SMALL view and the BIG view, but since SMALL goes first, the duplicates from BIG are "dupout-ed".

MACRO VARIABLES

This example gets us back to the macro language.

code: macro variables

```
/* look-up 31, unique markers in macro variables */
%let uniqueMarker=%sysfunc(datetime(),hex16.);
%put &=uniqueMarker.;
data _null_;
  set WORK.small;
  call symputX(cats("_31_&uniqueMarker._",IDS),"I am here","G");
run;
%put _user_;
data WORK.RESULT31;
  set WORK.BIG;
  where symexist(cats("_31_&uniqueMarker._",ID));
run;
```

The &uniqueMarker. is just a technical macro variable, generated based on the execution timestamp, which ensures that created macro variables will be unique for a particular run. Both DATA steps use the CATS() function to create a macro variable name based on string constant "_31_&uniqueMarker._" and the value of the ID/IDS variable. For example, if the code was run on September 9th, 2024 at 12:34:56, the ID value 42 would create a macro variable named _31_41DE6ABB9C000000_42. The first DATA step creates a list of macro variables with arbitrary values, "I am here" in our case, which are stored in the SAS global macro variables table. The second DATA step uses the SYMEXIST() function which checks if a macro variable with a particular name exists in the SAS global symbol table. If the macro variable does exist it means that it was created in the first DATA step.

Use of the SYMEXIST() function is not the only option here, though it is the cleanest in the setup we have. We get a similar result with use of the SYMGET() function or the RESOLVE() function. The first function allows us to retrieve macro variable value on the DATA step level. A snippet like this:

code: symget

```
  where symget(cats("_31_&uniqueMarker._",ID)) is not null;
```

does the job when used in the WHERE clause (if used in the IF-subsetting condition it produces data errors for not existing variables). The second function gives us possibility to resolve all elements of macro language (not only macro variables) in a DATA step. A snippet like that:

code: resolve

```
  where resolve(cats("&","_31_&uniqueMarker._",ID)) = "I am here";
```

does the job but also produces tons of "Apparent symbolic reference not resolved" warnings. Though a bit clumsy in this particular setup, the RESOLVE() function is extremely powerful tool, interesting use cases can be found in [**Carpenter(2) 2018**](also look-up related) and [**Jablonski(2) 2023**].

GOING ABROAD - TRAVEL BROADENS THE MIND

Now let us play a bit with "foreign languages", R, Python, and Lua for a start. In all three cases we are going to use SAS procedures to get programming interfaces to the external language. In the first case it is, already introduced, the IML procedure, in the second, also well-known, the FCMP procedure, and the third has its own dedicated one, the LUA procedure.

Attempts to set communication between SAS and R have been discussed and presented in the user community, see for example [**Holland 2005**] or [**Wei 2012**], but the most comfortable way to go is with the help of the IML procedure. To communicate with R some additional setup is needed. First of all, the RLANG option has to be enabled (it is a run time option, so it has to be enabled in the SAS configuration file), the second option, R_HOME, indicates R home directory location.

code: talking in R

```
/* look-up 32, "foreign languages" - R */
/* Check if RLANG is on and set environment variable */
proc options option=RLANG; run;
options set=R_HOME="/path/to/R/R-4.3.1/";
PROC IML;
  /* note about R 4.3.0 and later: https://support.sas.com/kb/70/253.html */
  call ExportDataSetToR("WORK.BIG", "BIG");
  call ExportDataSetToR("WORK.small", "small");
  submit / R;
     RESULT32 <- BIG[BIG$id %in% small$ids,1:3]
     head(RESULT32)
     tail(RESULT32)
  endsubmit;
  call ImportDataSetFromR("WORK.RESULT32", "RESULT32");
QUIT;
proc print data=WORK.RESULT32;
run;
```

The SAS Support note mentioned in the comment above points to a HotFix dedicated for SAS and R later than 4.3.0. The CALL EXPORTDATASETTOR() (creates an R data frame from a SAS data set) and CALL IMPORTDATASETFROMR() (creates a SAS data set from an R data frame) give us seamless communication/data exchange between both tools. The SUBMIT code block contains R code doing data selection. The head and tail are just for results previewing. More details about how to play with R from the IML procedure level can be found in [**Bewerunge 2011**], [**Bulaienko 2016**], or [**Gilsen 2023**].

The "traveling" between SAS and R feels like traveling between countries in Schengen Area, both the configuration and data transfer effort are minimal. For Python it is more like "you need a visa", you have to put a bit more effort: you have to set up two environment variables, download two packages, and modify saspy.py configuration file.

code: talking to Python

```
/* look-up 33, "foreign languages" - Python */
/* Set environment variables: */
options set=MAS_M2PATH="C:/SAShome/SASFoundation/9.4/tkmas/sasmisc/mas2py.py";
options set=MAS_PYPATH="/path/to/Python/Python312/python.exe";
/* Install packages:
  python -m pip install pandas saspy
Setup SASPY.PY config file, located for example in:
  /path/to/Python/Python312/Lib/site-packages/saspy/sascfg.py
For local SAS on Windows set it like:
  default={"saspath":"C:/SASHome/SASFoundation/9.4",
           "encoding":"utf8",
           "java":"C:/SASHome/SASPrivateJavaRuntimeEnvironment/9.4/jre/bin/java"
           } */
```

```
PROC FCMP;
length libpath fileBIG filesmall fileresult33 $ 512 output $ 42;
libpath      = pathname('WORK');
fileBIG      = catx("/", libpath, 'big.sas7bdat');
filesmall    = catx("/", libpath, 'small.sas7bdat');
fileresult33 = catx("/", libpath, 'result33.csv');

declare object py(python);
submit into py;
def lookup33(BIGpath, smallpath, CSVpath, libpath):
  """Output: outputKey"""
  import pandas
  import saspy

  #/* read data sets to pandas data frames */
  BIGdf = pandas.read_sas(BIGpath)
  smalldf = pandas.read_sas(smallpath)
  RESULT33 = BIGdf.merge(smalldf, left_on='id', right_on='ids', how='inner')
  #/* create CSV file for import */
  RESULT33[["id","date","value"]].to_csv(CSVpath, index=False)
  #/* create data set with SASPY */
  sas = saspy.SASsession(cfgname='default')
  sas.saslib(libref = 'out', path = libpath)
  dataSet = sas.dataframe2sasdata(
    df = RESULT33[["id","date","value"]],
    libref = 'out',
    table = 'RESULT33')
  sas.endsas()
  return dataSet.table
endsubmit;
rc = py.publish();
rc = py.call('lookup33', fileBIG, filesmall, fileresult33, libpath);
output = py.results['outputKey'];
put "output:" output;
RUN;

proc import
  file="%sysfunc(pathname(WORK))/result33.csv"
  out=csv_version_of_result33
  dbms=csv replace;
run;
proc print data=csv_version_of_result33;
run;
proc print data=RESULT33;
run;
```

When the configuration is ready, we create some temporary variables pointing to the input data location. Communication between SAS and `Python` is done with help of the so-called python object. The submit block contains `Python` code, you have to remember about indentations and that SAS comments are not honored, so to comment out code you have to use the # symbol. The `pandas` package is used to read SAS data sets into data frames. Next, the `pandas` data frame's `merge` method sub-selects data. The result is created in two ways. One, just a `csv` file is created and then the `IMPORT` procedure makes it a SAS data set. Two, the `saspy` package is used to create a SAS data set directly. More reading about the "house Slytherin" can be found in [**Lankham & Slaughter 2023**]. For the sake of a complete picture, the `FCMP` procedure is not the only interface to `Python` language available in SAS ecosystem. If you happen to have access to the SAS Viya platform, you are able to use the `PYTHON` procedure which makes things much easier. See [**Box 2023**] for details.

Staying in the traveling parallel, going between SAS and Lua would be like traveling between states, no effort required, the only thing to remember is that rules change between states.

code: talking in Lua

```
Proc LUA restart;
submit;
  --- /* get data from SMALL */
  local small = {}
  local dsid = sas.open("WORK.small")
  for row in sas.rows(dsid) do
    for n,v in pairs(row) do
      if n == "ids" then
        small[#small+1] = v
      end;
    end
  end
  sas.close(dsid)
  print(table.tostring(small))
  print(type(small))

  --- /* load BIG */
  local BIG = sas.read_ds("WORK.BIG")
    print(type(BIG))
  local result34 = {}
    print(table.tostring(result34))
  local cnt = 0
  local find = 0
  --- /* loop over rows of BIG and ... */
  for _,rowBIG in ipairs(BIG) do
    cnt = cnt + 1
    --- /* ... check if value of ID is in small */
    if (table.contains(small, rowBIG.id)) then
      find = find + 1
      vars = {}
        vars.id=rowBIG.id
        vars.date=rowBIG.date
        vars.value=rowBIG.value
      result34[#result34+1] = vars
    end
  end;
  print ("cnt=", cnt)
  print ("find=", find)

  -- /*write LUA table to SAS data set */
  sas.write_ds(result34, "work.result34")

  print(table.tostring(result34))
endsubmit;
run;
proc print data=RESULT34;
  format date yymmdd10.;
run;
```

Inside the LUA procedure we use the submit block to provide Lua code. SAS provides a bunch of utility functions out of the box to help us in moving data between SAS data sets and Lua tables, back and forth. Similarly to Python in the FCMP procedure, SAS comments are not respected, a double dash (--) has to be used. An introduction to SAS and Lua cooperation can be found in the following articles [**Tomas 2015**], [**Hu 2016**], [**Vijayaraghavan 2017**], and [**Khorlo 2019**].

MY PRIVATE PHONE-BOOK

Use of SAS indexes can significantly increase the processing speed of sub-setting data sets. The simplest experiment confirming the fact is to run one of the first examples (those "naive" ones) which uses the WHERE clause like for example this one:

code: WHERE clause

```
where ID in (1 2 3 6 13 17 42 101 303 555 9999);
```

If the filtered data set is indexed and we are sub-setting a small amount of data (a rule of thumb says: less then 5%), the process takes much less time. Additionally the MSGLEVEL= option, when set to i, shows a log info about index use. Though SAS indexes are very practical, sometimes with a specific data setup or filtering condition, their out of the box version does not solve the issue. Interesting discussion about such cases can be found in [**Keintz 2009**] or [**Jablonski 2019**]. The next snippet we discuss is inspired by the first of those two references.

code: user made index

```
/* look-up 35, SAS indexes and user-defined indexes */
data work.userDefinedIndex(
      index=(key)
      keep=key start end dataSet);
  set WORK.BIG /* WORK.BIG2 WORK.BIG3 ... - works with multiple data sets */
      curobs=curobs indsname=indsname end=eof;
  lag_co   = lag(curobs);
  lag_INDS = lag(indsname);
  lag_ID   = lag(id);
  newDS    = indsname NE lag_INDS;
  newBY    = id NE lag_ID;

  if EOF OR ((newDS OR newBY) AND 1 < _N_) then
    do;
      key     = coalesce(lag_ID,ID);
      dataSet = coalescec(lag_INDS,indsname);
      start   = (start<>1);
      end     = coalesce(lag_co,curobs);
      output userDefinedIndex;
      start=.;
      start+curobs;
    end;
run;
/* proc print data=userDefinedIndex(firstobs=12340 obs=12350); run; */
proc sql noprint;
  select IDS
  into :IDS separated by " "
  from work.small;
quit;
/* options msglevel=i; */
data _null_;
  call execute("data RESULT35; set");
  do until(eof);
    set work.userDefinedIndex end=EOF;
    where KEY in (&IDS.);
    call execute(catx(" ",dataSet,"(firstobs=",start,"obs=",end,")"));
  end;
  call execute("open=defer;run;");
stop;
run;
```

We use the fact that the BIG data set is sorted by the `ID` variable. Based on that, in the snippet we create additional data set which works for us like an index telling us which `ID`s occupy which observations. Further more, this user hand-made index can combine information on data from multiple data sets (as the snippet shows). We additionally put an index on the `ID` variable in the user-index data set . Then we use our small data set for indirect selection from the user hand-made index data set. In the final stage those sub-select observations of the user-index data set allow us to create the final, data-selecting, DATA step. The log looks like this:

the log - result

```
data RESULT35;
  set
    WORK.BIG(firstobs=1 obs=4) WORK.BIG(firstobs=5 obs=9)
    WORK.BIG(firstobs=10 obs=11) WORK.BIG(firstobs=20 obs=23)
    WORK.BIG(firstobs=52 obs=56) WORK.BIG(firstobs=70 obs=73)
    WORK.BIG(firstobs=183 obs=186) WORK.BIG(firstobs=455 obs=459)
    WORK.BIG(firstobs=1375 obs=1379) WORK.BIG(firstobs=2514 obs=2518)
    WORK.BIG(firstobs=45011 obs=45014)
  open=defer;
run;
```

&&&&&&&

When working with macro programming the indirect reference technique (already mentioned earlier) is considered to be an advance topic. Even though considered advanced, examples of code using double ampersand (`&&`) are very popular, even three (`&&&`) can be found easily. Much rarer case is the one where four (`&&&&`) or more are used, and the use is not a symptom of *macroitis* disease (see [**Schrempf 1995**]) but is a justified case.

An exercise like: *"Having the following setup:*

code: indirect referencing - exercise setup

```
%let VeryImportantMessage=This, is, PharmaSUG!!!;
%let T1=Very;
%let T2=Important;
%let T3=Message;
%let letter=T;
%let one=1;
%let two=2;
%let three=3;
```

print out the 'This, is, PharmaSUG!!!' text in the log, but using only*(!) macro variables: `letter`, `one`, `two`, and `three` in your code"*, that can by the way be resolved with the following line of code:

code: indirect referencing - the answer

```
%put
  &&&&&&&letter&one&&&letter&two&&&letter&three
;
```

happens *almost exclusively* as an academic exercise! (see Appendix D - "sevenfold" indirect referencing for a visual explanation, read [**Gerlach 1997**], [**Molter 2004**], and [**Matise 2015**] for understanding) That is why the next snippet can be considered interesting.

The code, like the previous one, also relies on the fact that the data are sorted (interesting read about different approaches to sorted data sets can be found in [**Lavery 2013**]).

code: quadruple indirect referencing

```
/* look-up 36, SAS macro indirect referencing */
data _null_;
  set WORK.BIG curobs=co;
  by ID;
  if first.ID then call symputX(cats("_st36_",ID),co,"G");
  if last.ID  then call symputX(cats("_en36_",ID),co,"G");
run;
data _null_;
  set WORK.small end=eof;
  call symputX(cats("_i36_",_N_),IDS,"G");
  if eof then call symputX("_n36_",_N_,"G");
run;

%macro lookup36();
data RESULT36;
  set
    %local i;
    %do i = 1 %to &_n36_.;
       WORK.BIG(firstobs=&&&&_st36_&&_i36_&i obs=&&&&_en36_&&_i36_&i)
    %end;
  open=defer;
run;
%mend lookup36;

options mprint symbolgen mlogic;
%lookup36()
options nomprint nosymbolgen nomlogic;
```

Because we want to use a similar trick to the one we just used, i.e. selection by observation number, the first DATA step goes through the BIG data set and for every value of the `ID` variable creates two macro variables: the `_st36_*` contains the observation number for a group start, the `_en36_*` contains the observation number for a group end. The second DATA step creates eleven macro variables: `_i36_1`, `_i36_2`, ..., `_i36_11` containing the `IDS` variable values we want to look-up in the BIG data set, and `_n36_` = 11 macro variable.

The `%lookup36()` macro loops from one to eleven. In the eleventh `%DO-LOOP` iteration (`i=11`, `IDS=9999`), for example for `&&&&_st36_&&_i36_&i`, we have:

- after the first macro processor pass: `&&_st36_&_i36_11`
- after the second macro processor pass: `&_st36_9999`, and
- after the third macro processor pass: `45011`

so eventually the 4GL we produce became a very familiar looking:

the log - result

```
WORK.BIG(firstobs=1 obs=4) WORK.BIG(firstobs=5 obs=9)
WORK.BIG(firstobs=10 obs=11) WORK.BIG(firstobs=20 obs=23)
WORK.BIG(firstobs=52 obs=56) WORK.BIG(firstobs=70 obs=73)
WORK.BIG(firstobs=183 obs=186) WORK.BIG(firstobs=455 obs=459)
WORK.BIG(firstobs=1375 obs=1379) WORK.BIG(firstobs=2514 obs=2518)
WORK.BIG(firstobs=45011 obs=45014)
```

WE HAVE TO GO DEEPER

In an early SEUGI[5] conference paper A. D. Forbes discusses a PROC FUNCTN procedure, see [**Forbes 1984**]. According to the article, in a nutshell, the procedure takes a data set of $(argument, value)$ pairs that represents a function in *mathematical sense* and (as the name suggests) creates a table look-up function, which use binary search over arguments and returns corresponding value. In our setup we could consider the following approach:

code: attempt to use PROC FUNCTN

```
DATA work.functionData;
  set work.small;
  rename IDS=arg;
  value=1;
run;
PROC SORT data=work.functionData;
  BY arg;
run;

PROC FUNCTN
  DATA=work.functionData
  NAME=myFunc
  DDNAME=work
  TYPE=1
;
  ID value;
  VAR arg;
run;

DATA work.RESULT37;
  set work.BIG;
  if myFunc(ID);
run;
```

Unfortunately for us, in the SAS9 system the result printed in the log is very unsatisfactory:

the log - unsatisfactory result

```
...

11   PROC FUNCTN
12     DATA=work.functionData
ERROR: Procedure FUNCTN not found.
13     NAME=myFunc
14     DDNAME=work
15     TYPE=1
16   ;
17     ID value;
18     VAR arg;
19   run;
NOTE: The SAS System stopped processing this step because of errors.

...
```

As we have already seen, when talking about the FCMP procedure, the use of data set and a binary search of a sorted list of values is implementable. But... there is one more way to implement a binary search over a given list of sorted values. Such a search can be implemented as a nested set of IF-THEN-ELSE statements, in our case of the form:

[5] SEUGI was the European version of SUGI, held, accroding to lexjansen.com, from 1983 through 2003.

code: fixed binary search

```
data work.RESULT37;
  set work.BIG;
    if (ID < 17) then
      do;
        if (ID < 3) then
          do;
            return=( ID=1 | ID=2 );
          end;
        else if (ID = 3) then return=(1);
        else if (ID > 3) then
          do;
            return=( ID=6 | ID=13 );
          end;
      end;
    else if (ID = 17) then return=(1);
    else if (ID > 17) then
      do;
        if (ID < 303) then
          do;
            return=( ID=42 | ID=101 );
          end;
        else if (ID = 303) then return=(1);
        else if (ID > 303) then
          do;
            return=( ID=555 | ID=9999 );
          end;
      end;
  if return;
run;
```

How we could write such code if our values list is a dynamically changing one? Well, the answer is pretty obvious. The SAS language component responsible for the language "dynamicity" is the macro language. We have seen a few simple, though very practical, macros that used the `%DO-LOOP` iterative processing. The example which generated the above snippet is an opportunity not only to discuss `%DO-LOOP`s, but also:

(1) positional and key-value macro parameters,
(2) macro language options (`MINOPERATOR` and `MPRINT`)
(3) conditional processing (`%IF-THEN-ELSE` statements),
(4) local and global variable scope (`%LOCAL` statement),
(5) macro functions (`%SCAN()`),
(6) macro variable arithmetic (`%EVAL()` macro function),
(7) use of 4GL functions on a macro programming level (`%SYSFUNC()` macro function),
(8) recursion (call to `%binSrch()` inside `%binSrch` definition, "Inception"), and
(9) special characters masking (aka. macro quoting, `%STR()` and `%SUPERQ()` macro functions).

code: SAS macro recursion for binary search

```
%macro binSrch(var,list,sep==)/minoperator;
%local l h i j v;
%let l=1;
%let h=%sysfunc(countw(%superq(list),%str( )));
%let i = %eval((&l.+&h.)/2);
%let v = %scan(&list.,&i.,%str( ));

%if &h. = 1  %then
  %do;
    return&sep.( &var.=&list. );
  %end;
%else %if &h. = 2 %then
  %do;
    return&sep.(&var.=%scan(&list.,1,%str( )) | &var.=%scan(&list.,-1,%str( )));
  %end;
%else %if NOT %sysevalf(%superq(v)=,boolean) %then
  %do;
    if (&var. < &v.) then
      do;
        %binSrch(&var.
                ,%do j=&l. %to %eval(&i-1); %scan(&list.,&j.,%str( )) %end;
                ,sep=&sep.)
      end;
    else if (&var. = &v.) then return&sep.(1);
    else if (&var. > &v.) then
      do;
        %binSrch(&var.
                ,%do j=%eval(&i+1) %to &h.; %scan(&list.,&j.,%str( )) %end;
                ,sep=&sep.)
      end;
  %end;
%mend binSrch;

options mprint;
data work.RESULT37;
  set work.BIG;
  %binSrch(ID, 1 2 3 6 13 17 42 101 303 555 9999)
  if return;
run;
```

When the `sep=` parameter of the macro is set to missing, then a call to the macro can be also embedded inside the `FCMP` procedure function definition. Other examples of recursive macros can be found in [**Adams 2003**], [**Itoh 2003**], [**Chung 2004**], or [**Watts 2010**]. As already mentioned, the macro language is exceptionally well explained in [**Carpenter 2016**], but three more articles discussing macro quoting are "must read" for a SAS professional, they are: [**O'Connor 1999**](!), [**Whitlock 2010**], and [**Chung & King 2009**].

SCALING OUT THE MERGE

To have a nice closure over the SAS syntax we return to merging data. We already wrote that the `MERGE` statement is a classic SAS way of doing table look-ups. Assumption about the merging is that all data sets taking part in the process have to be sorted by the `BY` variables. But if we decide to use different SAS library engine, i.e. the `SPDE` engine (introduced with SAS version 9), we do not have to worry about sorting. The *SPDE* stands for Scalable Performance Data Engine and it was design to improve work with big data sets by spreading the data across multiple disk drives to improve performance.

code: SPDE engine

```
/* MERGE with SPDE - Scalable Performance Data Engine */
options dlcreatedir;
libname s "%sysfunc(pathname(WORK))/SPDE";
libname s SPDE "%sysfunc(pathname(WORK))/SPDE"; /* the simplest form */
libname s list;

data S.BIG;
  set PUB.BIG;
  format date yymmdd10. value dollar10.2;
run;

proc copy in=work out=s;
  select small;
run;

data S.RESULT6B;
  MERGE
    S.BIG(in=B)
    S.small(in=S RENAME=(IDS=ID));
  BY ID;
  if S and B;
run;
```

The first step was to assign the SPDE library, with help of the DLCREATEDIR option we used sub-directory of the WORK in the example. In next step we copied both data sets to the S library. For SMALL, we used the COPY procedure. Notice we used original database library PUB as source, so the BIG data set is not-sorted. But the code for merging, except for library reference to S, looks exactly the same like in case of the standard SAS engine. Thanks to its properties the SPDE allows for "not-sorted" merging. The SPDE also handles indexes in a different way than "standard" SAS library, but to learn more about that we recommend reading [**Clifford 2004**] and [**Lavery & Whitlock 2006**] articles.

DIVE WITH DATA INTO THE VIYA-VERSE

In this section[6] we will refocus our linguistic target to SAS Viya, a platform introduced in 2016. Viya combines the power of SAS9 engine (i.e., BASE engine, referred there as *SPRE*) with in-memory processing executed through Cloud Analytics Services (CAS) engine. Viya introduces a lot of new extensions to the "good old" DATA step processing, e.g., in-memory parallel processing, it also introduce new language CAS-L (or maybe we should call it a dialect?)

First step to start work with CAS engine is to start CAS session. By default, when we log-in ourselves to Viya's SAS Studio, only the SPRE (SAS Programming Runtime Environment, aka. BASE SAS) session is started. The following snippet[7] starts CAS session called CASAUTO with session's default TIMEOUT= extended to 24 hours. In the second line, a default in-memory CAS library called casuser is assigned to SAS libraries under a caswork label.

code: start CAS session

```
cas CASAUTO sessopts=(timeout=86400);
libname caswork CAS caslib=casuser;
/* cas CASAUTO terminate; */ /* run to kill CAS session */
```

[6] Title inspired by David Weik's YouTube series: https://www.youtube.com/playlist?list=PLncvHGGelzhX1hMwfEHA2eBQOFgV1FfSZ

[7] All Viya related code snippets were executed on the *SAS Viya for Learners* environment (VLE, https://vle.sas.com/) which, in contrast to *SAS On Demand for Academics* (SODA), is accessible only to students with emails in the "edu" domain (SAS Institute, why?) The VLE also does not reveal the "full" SAS Viya potential.

CAS session can be connected to a particular server under particular port by setting up HOST= and PORT= parameters (the documentation covers dozens of options more).

FROM DISK TO MEMORY

To use potential of the CAS engine we have to have our data loaded into memory. To do it, the PROC CASUTIL utility procedure seems to be the best solution.

code: loading data into memory

```
proc casutil;
  load data=WORK.BIG   casout="BIG"      outcaslib="casuser" replace;
  load data=WORK.small casout="small"    outcaslib="casuser" replace;
  load data=WORK.small casout="small_ID" outcaslib="casuser" replace;

  list tables;
run;
```

The second copy of "in-memory" SMALL, renamed to SMALL_ID, will be used later.

IT LOOKS JUST THE SAME...

For many SAS programmers a "transition from SAS to Viya" seems to be scary process. But when we take a look at the following DATA step (that is executed in CAS!), at the first sight we cannot tell it runs in CAS. It look no different than standard BASE SAS DATA step program.

code: in-memory & parallel data step

```
data caswork.RESULT38;
  merge caswork.BIG(in=B) caswork.small(in=S rename=(ids=id));
  by id;
  if B AND S;
  t = _THREADID_;
run;
proc print data=caswork.RESULT38;
run;
```

A DATA step executed in CAS is said to be "*CAS enabled*". The same we say about procedures. The indicator of the fact that we moved to CAS processing is the following note in the log:

the log - CAS processing note

```
NOTE: Running DATA step in Cloud Analytic Services.[8]
```

A "supporting" indicator for parallel DATA step processing is the _THREADID_ system variable which contains thread number. Additional feature of CAS in-memory processing is that we do not have to sort those in-memory data sets (tables in CAS nomenclature) for BY-group processing! Introductory reading about those "new" features can be found in [**Secosky 2017**]. And a "general" introduction to the subject, with a very rich references page, can be found in [**Russell & Nelson 2018**].

MORE THAN JUST A UTILITY

The PROC CASUTIL has a lot of useful features, not only can it load data sets into in-memory CAS tables, it also works other way round, and since there is WHERE="..." option we can filter!

code: saving with filter

```
proc sql noprint;
  select ids
  into :ids separated by " "
  from work.SMALL;
```

[8] "*If I won the lottery I wouldn't tell anyone, but there would be signs*" ;-) ;-) ;-)

```
quit;
proc casutil;
  save casdata="BIG" outcaslib="casuser" casout="RESULT39" replace
  WHERE="id in (&ids.)" /* a text string */
  ;
run;
```

When we look into the LOG we will see the following note:

the log - sashdata file

```
NOTE: Cloud Analytic Services saved the file RESULT39.sashdat in caslib CASUSER.
```

What is that `.sashdat` file? Well, the `sashdat` is a new file format used by CAS to store in-memory data on disk in an efficient, for future reads, way. So now, we are not surprised that:

the log - missing file

```
01   proc print data=caswork.RESULT39;
NOTE: The table CASWORK.RESULT39 does not exist in Cloud Analytic Services.
ERROR: File CASWORK.RESULT39.DATA does not exist.
02   run;
```

returns an error message. We basically filtered out in-memory table and automatically stored it on a disk drive as the `sashdat` file. To get it back to `caswork` library and print it we run:

code: print SASHDAT file

```
proc casutil;
  load incaslib="casuser" casdata="RESULT39.sashdat"
       casout="RESULT39" replace;
run;
proc print data=caswork.RESULT39;
run;
```

But be aware, it is not "symmetric"! Loading does not respect the `WHERE="..."` option. To have our loading to work correctly we have to use the data set option `WHERE=(...)`.

code: loading with filter

```
proc casutil; /* !!! does not work as expected */
  load data=WORK.BIG outcaslib="casuser" casout="RESULT40" replace
  WHERE="id in (&ids.)";
run;
proc casutil;
  load data=WORK.BIG(WHERE=(id in (&ids.)))
  outcaslib="casuser" casout="RESULT40" replace  ;
run;
```

The first snippet even returns a warning note:

the log - loading with filter

```
WARNING: The WHERE option is ignored when using the DATA= option
         in the LOAD statement.
```

A BRAND NEW DIALECT

With the SAS Viya a brand new programming language was introduced. The `CAS-L`, Cloud Analytic Services Language, is a SAS dialect dedicated native to CAS server *actions* processing. The `CAS-L` is a statement-based language and programs are built with 4GL-like statements, with so-called *actions*, or even user-defined functions. According to "CASL Programmer's Guide":

"*CAS actions are executable routines that the CAS server makes available to client programs. They are the smallest unit of work for the CAS server. There are actions for each of the many analytic algorithms, for data management, for administration, and for simple housekeeping. Actions are organized into groups called action sets. For example, the* `Table` *action set contains actions for loading tables, deleting tables, managing table attributes, and so on.*"

According to the CAS-L documentation[9] there are almost 570 actions, spanning across more than 135 action sets. Actions for CAS are like *LEGO* bricks, and every code executed on CAS is, under the hood, decomposed into actions. Even that CAS enabled DATA step. If we run :

code: listing CAS history

```
cas casauto listhistory _all_;
```

the LOG presents something similar to the following myriad of notes:

the log - DATA step in actions

```
NOTE: 01: action table.tableInfo /
            name='BIG', caslib='CASUSER', quiet=true;
NOTE: 02: action table.columnInfo /
            table={name='BIG', caslib='CASUSER'},
            extended=true, sastypes=false;
NOTE: 03: action table.tableInfo /
            name='SMALL', caslib='CASUSER', quiet=true;
NOTE: 04: action table.columnInfo /
            table={name='SMALL', caslib='CASUSER'},
            extended=true, sastypes=false;
NOTE: 05: action builtins.about;
NOTE: 06: action dataStep.checkBinary /
            msglevel='N', function={}, call_routine={},
            format={'YYMMDD', 'DOLLAR'}, informat={};
NOTE: 07: action sessionProp.getSessOpt /
            name='dataStepReplaceTable';
NOTE: 08: action sessionProp.setSessOpt /
            dataStepReplaceTable=true;
NOTE: 09: action dataStep.runBinary /
            nThreads=0, compilerEncoding='ansi', single='noinput',
            libname={{libname='CASWORK ', caslib='CASUSER'}};
NOTE: 10: action sessionProp.setSessOpt /
            dataStepReplaceTable=true;
```

So, this is how CAS programming looks in action! There is a lot to learn. There is a lot of material for creating a training. But we will focus ourselves on a few selected examples that works for our purpose - table look-up. The rest is a homework...

I DID NOT KNOW THESE TYPES

CAS-L introduced a bunch of new variable types, like for example dictionaries or arrays (both presented below). Since we added the `tbl.where` entry to the dictionary, the `TABLE.FETCH` action will use it for data selection in the `TABLE=` option. The `vars` variable, containing an array of variables names, will be used in the `FETCHVARS=` option. We can also reorganize sorting order before saving `result=` in `RESULT41` variable, so we could save it later as a CAS table in the `casuser` CAS-library.

[9] As of February 2025, https://go.documentation.sas.com/doc/en/pgmsascdc/v_055/allprodsactions/actionsByName.htm

code: new data types and action sets

```
proc cas;
  session casauto;

  tbl.name = "BIG";
  tbl.where = "id in (1 2 3 6 13 17 42 101 303 555 9999)";

  describe tbl;
  print "dictionary: " tbl;

  vars = {"id", "date", "value"};

  describe vars;
  print "array: " vars;

  table.fetch result=RESULT41 /
    table = tbl
    fetchvars = vars
    sortby = {
      {name="id",   order="ascending"},
      {name="date", order="descending"}
    }
    to = &sysmaxlong.
    index = FALSE
  ;

  describe RESULT41; /* see the structure */
  print RESULT41;

  saveresult RESULT41 CASLIB="casuser" CASOUT="RESULT41" REPLACE;
quit;
```

The LOG notes after execution show the new structures:

the log - new data types and action sets

```
NOTE: Active Session now casauto.

dictionary ( 2 entries, 2 used);
 [name] string;
 [where] string;
dictionary: name=BIG,where=id in (1 2 3 6 13 17 42 101 303 555 9999)

array ( 3 entries, 3 used);
 [ 1] string;
 [ 2] string;
 [ 3] string;
array: id,date,value

dictionary ( 1 entries, 1 used);
 [Fetch] Table ( [47] Rows [3] columns
Column Names:
 [1] id              [               ] (double)
 [2] date            [               ] (double) [YYMMDD10.]
 [3] value           [               ] (double) [DOLLAR10.2]
NOTE: The table RESULT41 loaded into casuser with 47 observations
      and 3 variables.
```

CAMERA, ROLL, ACTION!

We already took a look at the TABLE action set, in this section we investigate it further and continue with two more action sets.

In the first snippet, with the COLUMNS= option, we rename variable ids to id in the SMALL_ID table (it is in-memory, so yes, table) so we could use it later for filtering data.

code: CAS action set: alterTable

```
proc cas;
  session casauto;

  table.alterTable /
    name="SMALL_ID"
    columns = {{name="ids", reName="id"}}
  ;
quit;
```

In the second one, we use the WHERETABLE= option to provide a table which will be used for data filtering by the TABLE.FETCH action.

code: CAS action set: table

```
proc cas;
  session casauto;

  table.fetch result=RESULT42 /
    table = {name="BIG",
             whereTable = {name="SMALL_ID",
                           vars={{name="id"}}
                          } /* filtering table */
            },
    fetchvars = {"id", "date", "value"},
    sortby={
      {name="id",   order="ascending"},
      {name="date", order="ascending"}
    },
    to=&sysmaxlong.,
    index=FALSE;

  saveresult RESULT42 CASLIB="casuser" CASOUT="RESULT42" REPLACE;
quit;
```

The next action set, the *DATA Step Action Set*, looks similar to the IML procedure's SUBMIT block, though this one executes code stored in a text string.

code: CAS action set - dataStep

```
proc cas;
  session casauto;

  c.BIG    = "casuser.BIG(in=B)";
  c.small  = "casuser.small(in=S rename=(ids=id))";
  c.cond   = "if B AND S;";
  c.result = "casuser.RESULT43";
  describe c;
  codetext = "data "  || c.result || ";" ||
             "merge " || c.BIG    || " " || c.small || ";" ||
             "by id;" || c.cond   || "t = _THREADID_; run;";
  describe codetext;

  call streaminit(43);
```

```
  dataStep.runCode /
    nThreads = rand("integer",2,5), /* random number of threads */
    code = codetext;
  run;
quit;
```

The RUNCODE action executes text string, provided through the CODE= option, as if it was regular DATA step code. The RUNCODE action has a sibling, the RUNCODETABLE action, that runs DATA step code stored in a CAS table. Additionally, by setting the NTHREADS= option, we can determine the number of threads we want to run our code on. Worth remembering is that double exclamation mark does not work here as concatenation operator anymore.

The *FedSQL Action Set* works similarly to the DATA Step Action Set, though in this case instead DATA step code we are using an SQL query. Joining tables through FEDSQL.EXECDIRECT action works almost the same, but we have some more diagnostics options we can use to see the execution plan, methods used, etc.

code: CAS action set - fedSQL

```
proc cas;
  session casauto;
  fedSQL.execDirect /
    query = "create table CASUSER.RESULT44 {options replace=true} as
             select b.*
             from
              CASUSER.BIG as b
             inner join
              CASUSER.SMALL as s
             on b.id = s.ids"
    method = TRUE
    showPlan = TRUE
    showStages = TRUE;
quit;
```

Just to be clear we barely scratched the surface of CAS-L actions programming and there is still a ton of things to learn. Hopefully we gave you a "good enough glimpse" to motivate you to learning.

At the beginning of the CAS-L section we mentioned that we have various statements and even user-defined functions at our disposal. This next example presents some of them.

code: CAS-L statements and functions

```
/* Statements and user-defined functions */
proc cas;
  session casauto;
  function tableLookUp(BIG,small);
    action table.tableinfo result=snr / table=small; /* get some metadata */
    action table.tableinfo result=Bnr / table=BIG;
    snr = snr.TableInfo[1,"Rows"];
    Bnr = Bnr.TableInfo[1,"Rows"];

    table.fetch result=s /  /* fetch small data into s */
      table=small
      maxRows=snr, from=1, to=snr,
      index=FALSE;

    table.fetch result=B / /* fetch BIG data into B */
      table=BIG,
      maxRows=Bnr, from=1, to=Bnr,
      index=FALSE;
```

```
    table.fetch result=RESULT45 / /* get a "template" for result */
      table=BIG,
      maxRows=0,
      index=FALSE;

    s=s.Fetch;
    B=B.Fetch;
    RESULT45=RESULT45.Fetch;

    describe s;
    describe B;
    describe RESULT45;

    a={};
    do r over s; /* loop to put data from s to an array a */
      print ">s>" r.ids;
      a = a + {r.ids};
    end;

    describe a;
    print ">a>" a;

    do r over B; /* loop over data from B */
      if r.id in a then  /* check if ID exist in the array a */
        do;
          print ">B>" r.id put(r.date, date11.) put(r.value, dollar10.2);
          addRow(RESULT45, r); /* append row to result */
        end;
    end;

    return(RESULT45);
  end func;
  RESULT45 = tableLookUp("BIG","small"); /* call the function */

  describe RESULT45;
  print RESULT45;

  saveresult RESULT45 CASLIB="casuser" CASOUT="RESULT45" REPLACE;
run;

  describe tableLookUp; /* type is "function" */
quit;
```

In this pretty long snippet we defined CAS-L function `tableLookUp` with two arguments. With help of the `TABLE.TABLEINFO` action we extracted number of rows from both tables, we fetch BIG and SMALL into tables `B` and `s`. With the `DO OVER` loop we took data from `s` and stored them in array `a`. The second `DO OVER` loop iterated over every row in `B` and we checked if given value of `ID` exists in `a`, and if so then we added that row to result. For more CAS-L related reading see [**Sober & Kinnebrew 2020**] and be sure to see article's GitHub repository!

One thing to remember(!) at the end - the `CASUSER` CAS-library is an in-memory one, so if we do not want our all results "*to perish in void of in-memory address space*", we have to save all those results on disk, either as `.sas7bdats` or as `.sashdats` or in some other format (there are plenty).

THAT (+3) EXTRA

As we have already seen SAS is a very flexible and versatile tool, and provides a very wide range of different programming approaches to table look-ups. But wait, that is not all. The additional three examples present functionality called SAS Packages introduced in [**Jablonski 2020**] and further discussed for example in [**Jablonski 2021**], see also recording [**Jablonski 2023(video)**]. For an introduction about how to install and use the SAS Packages Framework and packages see the Appendix B - install the SPF and packages. When the SAS Packages Framework (SPF) and packages are ready for use, run the following snippet to enable SPF:

code: sas packages framework

```
/* enable SPF */
filename packages "/path/to/SAS/PACKAGES";
%include packages (SPFinit.sas);
```

Having the SAS Packages Framework ready we can look at examples. The first two are variations about the naive approach we discussed already, the third shows how macros can extend the power of the DATA step as a data processing tool.

THE BASEPLUS PACKAGE

The first one uses the `%minclude()` macro from the `basePlus` package. The macro is a workaround for the `%INCLUDE` statement's limitation, i.e., the fact that only valid SAS statements can be included. The `%minclude()` macro allows including arbitrary text from a file into SAS code. In this case the "1 2 3 6 13 17 42 101 303 555 9999" text string is taken literally from the `small.txt` file and is included inside the `WHERE` statement.

code: packages 1

```
/* look-up 97, naive selection, with basePlus */
%loadPackage(basePlus)
filename f97 "%workPath()/small.txt";
data WORK.RESULT97;
  set WORK.BIG;
  WHERE id in (%minclude(f97));
run;
filename f97 clear;
```

The `%minclude()` macro from the `basePlus` package, though implemented using a different approach, was influenced by the "Embedding any code anywhere into SAS programs" blog post by Leonid Batkhan (`https://blogs.sas.com/content/sgf/2023/05/30/`).

THE MACROARRAY PACKAGE

The second example uses the `macroArray` package which is designed to make work with macro variable arrays (MVAs) easier and more convenient. In the code, first the `%ARRAY()` macro is used to create a macro variable array named `IDS98_` directly from the SMALL data set, and next the `%DO_OVER()` macro retrieves and pastes values of the MVA into the `WHERE` statement.

code: packages 2

```
/* look-up 98, naive selection, with Macro Variable Arrays */
%loadPackage(macroArray)
%array(ds=WORK.small,vars=IDS|IDS98_,macarray=Y)
%put %IDS98_(1) %IDS98_(5) %IDS98_(11);
data WORK.RESULT98;
  set WORK.BIG;
  WHERE id in (%do_over(IDS98_));
run;
```

The `macroArray` package, though implemented from scratch, was influenced by works of Ted Clay and David Katz (see [**Clay 2004**] and [**Clay 2006**]), and itself is described in details in [**Jablonski 2024**].

THE SQLINDS PACKAGE

The third example shows that from a programmer point of view, it is not only the DS2 procedure which allows us to "have" an SQL query as a data source in a DATA step. The SQLinDS package utilizes macros, user-defined functions, and SQL views under the hood, but thanks to the form the solution is provided, i.e. a SAS package, from a user perspective at the end everything reduces to writing %SQL(<here goes my query>). And it works!

code: packages 3

```
/* look-up 99, SQLinDS - Macro Function Sandwich approach */
%loadPackage(SQLinDS)
data WORK.RESULT99;
  SET %SQL(select B.* from WORK.BIG as B,WORK.small as s where B.ID = s.IDS);
run;
```

The SQLinDS package, though fully developed and production usage ready, was developed as a hobby project being a tribute to Mike Rhoads' macro-function-sandwich idea presented in [**Rhoads 2012**].

A BIT OF SUMMARY

At this point we have 63 (+3) data sets (named ****.RESULT**) with observations selected from the BIG data set. We can say that we successfully finished the first step of the process, i.e. the data selection. Usually at this point further summary of the data is done. We execute the aggregation by running the next snippet that randomly selects one of those 63 (+3) data sets end executes a brief summary.

code: short summary

```
/* get list of data sets named "RESULT..." */
options obs=1;
data list;
  set
    WORK.RESULT:
    S.RESULT:
    CASWORK.RESULT:
  indsname=i;
  indsname=i;
  keep indsname;
run;
options obs=MAX;
proc sort data = list
  SORTSEQ=LINGUISTIC(NUMERIC_COLLATION=ON);
by i:;
run;
/* randomly select one of data sets */
data _null_;
    point = rand("integer",1,NOBS);
    set list POINT=point NOBS=NOBS;
    call symputX("EXAMPLE_DATA",indsname,"G");
    stop;
run;
/* aggregate data */
%put "Results from &EXAMPLE_DATA.";
title "Results from &EXAMPLE_DATA.";
proc tabulate data=&EXAMPLE_DATA.;
  class ID;
  var value;
  table id,value*(sum n mean);
run;
```

The TABULATE procedure is used here to generate the summary table which looks more or less like this:

ID	Sum	N	Mean
1	519.61	4	129.90
2	668.47	5	133.69
3	219.09	2	109.55
6	518.47	4	129.62
13	884.55	5	176.91
17	657.75	4	164.44
42	561.61	4	140.40
101	745.05	5	149.01
303	774.74	5	154.95
555	643.80	5	128.76
9999	648.60	4	162.15

Table 1: Summary for selected IDs

Of course we can easily assume, from the number of code snippets in previous sections, that this is not the only way we can aggregate our data. Except already used `PROC TABULATE` the list, among other, includes:

- `PROC MEANS`/`PROC SUMMARY`,
- `PROC UNIVARIATE`,
- `PROC SQL`,
- `DATA` Step with `BY`-group processing,
- `DATA` Step with `HASH` table,
- or maybe even the Aggregation Action Set.

So in fact in many cases the aggregation could be easily incorporated into the process at the first stage.

CONCLUSION

One obvious conclusion we can state is that SAS is a very syntax-rich and flexible language. In the article, we refreshed our memory about how the table look-up task can be solved and how many ways to do it are out there. Also we "experimentally proved" that you can be a lazy teacher but still provide a full SAS course to your students, even having just one exercise to solve for them. Hopefully you enjoyed the route 66.

The End

REFERENCES

As we think of these references as something of a historical collection, we ordered them by year, and then alphabetically by author within year. The links to each paper are hyperlinks (if a link is broken into multiple lines remember to copy all of it). For your convenience **all the articles** listed here were also collected in two places. Visit:

https://pages.mini.pw.edu.pl/~jablonskib/SASpublic/WUSS2024_125/

or

https://github.com/yabwon/WUSS2024_PAPER125/

to download them if you wish. Also a file with **all code snippets** presented in this article can be found in those locations.

ARTICLES AND BOOKS

[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

[Mays 1976] Steven Mays, "NON-STATISTICAL USES OF SAS", SUGI.ONE Proceedings, 1976,
https://support.sas.com/resources/papers/proceedings-archive/SUGI76/Sugi-76-16%20Mays.pdf

[Henderson 1982] Don Henderson, "SAS Tutorial: Table Lookup Techniques", SUGI Proceedings, 1982,
https://support.sas.com/resources/papers/proceedings-archive/SUGI82/Sugi-82-146%20Henderson.pdf

[Henderson 1983] Don Henderson, "The SAS Supervisor", SUGI Proceedings, 1983,
https://communities.sas.com/t5/SAS-Communities-Library/The-SAS-Supervisor/ta-p/429216

[Forbes 1984] A. D. Forbes, "A Procedure for Creating Table Look-Up Functions from SAS Data Sets", SEUGI Proceedings, 1984,
https://support.sas.com/resources/papers/proceedings-archive/SEUGI1984/
A%20Procedure%20for%20Creating%20Table%20Look-Up%20Functions%20from%20SAS%20Data%20Sets.pdf

[Lafler 1992] Kirk Paul Lafler, "Using the SQL Procedure", SUGI Proceedings, 1992,
https://support.sas.com/resources/papers/proceedings-archive/SUGI92/Sugi-92-90%20Lafler.pdf

[Scerbo 1992] Marge Scerbo, "Dataset Manipulation with Screen Control Language (SCL)", SUGI Proceedings, 1992,
https://support.sas.com/resources/papers/proceedings-archive/SUGI92/Sugi-92-41%20Scerbo.pdf

[Schrempf 1995] Margaret K. Schrempf, "Macroitis - A Virus or a Drug", SUGI Proceedings, 1995,
https://support.sas.com/resources/papers/proceedings-archive/SUGI95/Sugi-95-13%20Schrempf.pdf

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

[Gerlach 1997] John R. Gerlach, "The Six Ampersand Solution", NESUG Proceedings, 1997,
https://www.lexjansen.com/nesug/nesug97/posters/gerlach.pdf

[Whitlock 1997] Ian Whitlock, "A SAS Programmer's View of the of the SAS Supervisor", SUGI Proceedings, 1997,
https://support.sas.com/resources/papers/proceedings/proceedings/sugi22/ADVTUTOR/PAPER34.PDF

[Whitlock(2) 1997] Ian Whitlock, "CALL EXECUTE: How and Why", SUGI Proceedings, 1997,
https://support.sas.com/resources/papers/proceedings/proceedings/sugi22/CODERS/PAPER70.PDF

[Virgile 1997] Bob Virgile, "MAGIC WITH CALL EXECUTE", SUGI Proceedings, 1997,
https://support.sas.com/resources/papers/proceedings/proceedings/sugi22/CODERS/PAPER86.PDF

[Patton 1998] Nancy K. Patton, "IN & OUT of CNTL with PROC FORMAT", SUGI Proceedings, 1998,
https://support.sas.com/resources/papers/proceedings/proceedings/sugi23/Coders/p68.pdf

[Woolridge & Lau 1998] Greg M. Woolridge, Winnie Lau, "Happiness is Using Arrays to Make Your Job Easier",
PharmaSUG Proceedings, 1998, https://www.lexjansen.com/pharmasug/1998/TECH/WOOLRIDG.PDF

[Virgile 1998] Bob Virgile, "Introduction to Arrays", PharmaSUG Proceedings, 1998,
https://www.lexjansen.com/pharmasug/1998/DATA_VAL/VIRGILE1.PDF

[O'Connor 1999] Susan O'Connor, "Secrets of Macro Quoting Functions – How and Why", NESUG Proceedings, 1999,
https://www.lexjansen.com/nesug/nesug99/bt/bt185.pdf

[Shi & Zhang 1999] Jingren Shi, Shiling Zhang, "Tips about Using Data Step Option Point access",
MWSUG Proceedings, 1999, https://www.lexjansen.com/mwsug/1999/paper08.pdf

[Virgile 1999] Bob Virgile, "How MERGE Really Works", NESUG Proceedings, 1999,
https://www.lexjansen.com/nesug/nesug99/ad/ad155.pdf

[Aker 2000] Sandra Lynn Aker, "Using KEY= to Perform Table Look-up", SUGI Proceedings, 2000,
https://support.sas.com/resources/papers/proceedings/proceedings/sugi25/25/po/25p234.pdf

[Franklin & Jensen 2000] Gary Franklin, Art Jensen, "Integrity Constraints and Audit Trails Working Together",
PharmaSUG Proceedings, 2000, https://www.lexjansen.com/pharmasug/2000/DMandVis/dm18.pdf

[Riba 2000] S. David Riba, "How to Use the Data Step Debugger", SUGI Proceedings, 2000,
https://support.sas.com/resources/papers/proceedings/proceedings/sugi25/25/btu/25p052.pdf

[Carpenter 2001] Arthur L. Carpenter, "Table Lookups: From IF-THEN to Key-Indexing", SUGI Proceedings, 2001,
https://support.sas.com/resources/papers/proceedings/proceedings/sugi26/p158-26.pdf

[Dorfman 2001] Paul M. Dorfman, "Table Look-Up by Direct Addressing: Key-Indexing - Bitmapping - Hashingpdf document", SUGI Proceedings, 2001, https://support.sas.com/resources/papers/proceedings/proceedings/sugi26/p008-26.pdf

[Luo 2001] Haiping Luo, "That Mysterious Colon (:)", SUGI Proceedings, 2001, https://support.sas.com/resources/papers/proceedings/proceedings/sugi26/p073-26.pdf

[Shoemaker 2001] Jack Shoemaker, "Eight PROC FORMAT Gems", SUGI Proceedings, 2001, https://support.sas.com/resources/papers/proceedings/proceedings/sugi26/p062-26.pdf

[Carpenter 2002] Arthur L. Carpenter, "Building and Using Macro Libraries", SUGI Proceedings, 2002,
https://support.sas.com/resources/papers/proceedings/proceedings/sugi27/p017-27.pdf

[Dorfman & Snell 2002] Paul M. Dorfman, Gregg P. Snell, "Hashing Rehashed", SUGI Proceedings, 2002,
https://support.sas.com/resources/papers/proceedings/proceedings/sugi27/p012-27.pdf

[Keelan 2002] Stephen Keelan, "Off and Running with Arrays in SAS", NESUG Proceedings, 2002,
https://www.lexjansen.com/nesug/nesug02/bt/bt002.pdf

[Shoemaker 2002] Jack Shoemaker, "PROC FORMAT in Action", SUGI Proceedings, 2002,
https://support.sas.com/resources/papers/proceedings/proceedings/sugi27/p056-27.pdf

[Dorfman & Snell 2003] Paul M. Dorfman, Gregg P. Snell, "Hashing: Generations", SUGI Proceedings, 2003,
https://support.sas.com/resources/papers/proceedings/proceedings/sugi28/004-28.pdf

[Adams 2003] John H. Adams, "The power of recursive SAS macros - How can a simple macro do so much", SUGI Proceedings, 2003, https://support.sas.com/resources/papers/proceedings/proceedings/sugi28/087-28.pdf

[Fehd 2003] Ronald Fehd, "ARRAY: construction and usage of arrays of macro variables", NESUG Proceedings, 2003, https://www.lexjansen.com/nesug/nesug03/cc/cc015.pdf

[Itoh 2003] Yohji Itoh, "A Recursive SAS Macro Technique and its Application to Statistics", Conference: SUGI-J, 2003,
https://www.researchgate.net/publication/344314192_A_Recursive_SAS_Macro_Technique_and_its_Application_to_Statistics

[Chung 2004] Chang Y. Chung, "Recursive Subroutine Macros", NESUG Proceedings, 2004,
https://www.lexjansen.com/nesug/nesug04/po/po02.pdf

[Clay 2004] Ted Clay, "Macro Arrays Make %DO-Looping Easy", WUSS Proceedings, 2004,
https://www.lexjansen.com/wuss/2004/coders_corner/c_cc_macro_arrays_make_doloo.pdf

[Clifford 2004] Billy Clifford, "Scalable Access to SAS Data", SCSUG Proceedings, 2004,
https://www.lexjansen.com/scsug/2004/Clifford%20-%20Scalable%20Access%20to%20SAS%20Data.pdf

[Howard 2004] Neil Howard, "How SAS Thinks or Why the DATA Step Does What It Does", SUGI Proceedings, 2004,
https://support.sas.com/resources/papers/proceedings/proceedings/sugi29/252-29.pdf

[Molter 2004] Michael J. Molter, "The Role of Consecutive Ampersands in Macro Variable Resolution and the Mathematical Patterns that Follow", SUGI Proceedings, 2004,
https://support.sas.com/resources/papers/proceedings/proceedings/sugi29/063-29.pdf

[Karp 2004] Andrew H. Karp, "Steps to Success with PROC MEANS", SUGI Proceedings, 2004,
https://support.sas.com/resources/papers/proceedings/proceedings/sugi29/240-29.pdf

[Clifford 2005] Billy Clifford, "Frequently Asked Questions about SAS Indexes", SUGI Proceedings, 2005,
https://support.sas.com/resources/papers/proceedings/proceedings/sugi30/008-30.pdf

[Eason 2005] Jenine Eason, "Proc Format, a Speedy Alternative to Sort/Merge", SUGI Proceedings, 2005,
https://support.sas.com/resources/papers/proceedings/proceedings/sugi30/054-30.pdf

[First & Schudrowitz 2005] Steve First, Teresa Schudrowitz, "Arrays Made Easy: An Introduction to Arrays and Array Processing", SUGI Proceedings, 2005, https://support.sas.com/resources/papers/proceedings/proceedings/sugi30/242-30.pdf

[Holland 2005] Philip R. Holland, "SAS to R to SAS", PHUSE Proceedings, 2005,
https://www.lexjansen.com/phuse/2005/cc/cc03.pdf

[Suhr 2005] Diana Suhr, "Arrays by example", WUSS Proceedings, 2005,
https://www.lexjansen.com/wuss/2005/sas_essentials/ess_arrays_by_example.pdf

[Michel 2005] Denis Michel, "CALL EXECUTE: A Powerful Data Management Tool", SUGI Proceedings, 2005,
https://support.sas.com/resources/papers/proceedings/proceedings/sugi30/027-30.pdf

[Clay 2006] Ted Clay, "Five Easy (To Use) Macros", PNWSUG Proceedings, 2006,
https://www.lexjansen.com/pnwsug/2006/PN22TedClayFiveMacros.pdf

[Fickbohm 2006] David Fickbohm, "Using the DATASETS Procedure", SUGI Proceedings, 2006,
https://support.sas.com/resources/papers/proceedings/proceedings/sugi31/032-31.pdf

[Lavery & Whitlock 2006] Russ Lavery, Ian Whitlock, "An Annotated Guide: The New 9.1, Free & Fast SPDE Data Engine", NESUG Proceedings, 2006,
https://www.lexjansen.com/nesug/nesug06/io/io15.pdf

[Whitlock 2006] Ian Whitlock, "How to Think Through the SAS DATA Step", SUGI Proceedings, 2006,
https://support.sas.com/resources/papers/proceedings/proceedings/sugi31/246-31.pdf

[Fickbohm 2007] David Fickbohm, "Using the DATASETS Procedure Part II", SGF Proceedings, 2007,
https://support.sas.com/resources/papers/proceedings/proceedings/forum2007/070-2007.pdf

[Lavery 2007] Russell Lavery, "An Animated Guide: The Map of the SAS Macro Facility", PHUSE Proceedings, 2007,
https://www.lexjansen.com/phuse/2007/is/IS01.pdf

[Secosky 2007] Jason Secosky, "User-Written DATA Step Functions", SGF Proceedings, 2007,
https://support.sas.com/resources/papers/proceedings/proceedings/forum2007/008-2007.pdf

[Secosky & Bloom 2007] Jason Secosky, Janice Bloom, "Getting Started with the DATA Step Hash Object",
SGF Proceedings, 2007, http://www2.sas.com/proceedings/forum2007/271-2007.pdf

[Whitlock 2007] Marianne Whitlock, "The program Data Vector AS an Aid to DATA Step Reasoning", NESUG Proceedings, 2007,
https://www.lexjansen.com/nesug/nesug07/ff/ff20.pdf

[Wright 2007] Wendi L. Wright, "Creating a Format from Raw Data or a SAS Dataset", SGF Proceedings, 2007,
https://support.sas.com/resources/papers/proceedings/proceedings/forum2007/068-2007.pdf

[Bilenas 2008] Jonas V. Bilenas, "I Can Do That With PROC FORMAT", SGF Proceedings, 2008,
https://support.sas.com/resources/papers/proceedings/pdfs/sgf2008/174-2008.pdf

[Mack 2008] Curtis Mack, "MODIFY The Most Under-Appreciated of the Data Step File Handling Statements",
PNWSUG Proceedings, 2008, https://www.lexjansen.com/pnwsug/2008/CurtisMack-Modify.pdf

[Wright 2008] Wendi L. Wright, "PROC TABULATE and the Neat Things You Can Do With It", SGF Proceedings, 2008,
https://support.sas.com/resources/papers/proceedings/pdfs/sgf2008/264-2008.pdf

[Winn Jr. 2008] Thomas J. Winn Jr., "Introduction to PROC TABULATE", SGF Proceedings, 2008,
https://support.sas.com/resources/papers/proceedings/pdfs/sgf2008/171-2008.pdf

[Chung & King 2009] Chang Y. Chung, John King, "Is This Macro Parameter Blank?", SGF Proceedings, 2009,
https://support.sas.com/resources/papers/proceedings09/022-2009.pdf

[Dorfman 2009] Paul M. Dorfman, "From Obscurity to Utility: ADDR, PEEK, POKE as DATA Step Programming Tools",
SGF Proceedings, 2009, https://support.sas.com/resources/papers/proceedings09/010-2009.pdf

[Dorfman & Vyverman 2009] Paul M. Dorfman, Koen Vyverman, "The DOW-Loop Unrolled", SGF Proceedings, 2009,
https://support.sas.com/resources/papers/proceedings09/038-2009.pdf

[Eberhardt 2009] Peter Eberhardt, "A Cup of Coffee and Proc FCMP: I Cannot Function Without Them",
SGF Proceedings, 2009, https://support.sas.com/resources/papers/proceedings09/147-2009.pdf

[Keintz 2009] Mark Keintz, "A Faster Index for Sorted SAS Datasets", SGF Proceedings, 2009,
https://support.sas.com/resources/papers/proceedings09/024-2009.pdf

[Watts 2010] Perry Watts, "Using Recursion to Trace Lineages in the SAS ODS Styles.Default Template",
NESUG Proceedings, 2010, https://www.lexjansen.com/nesug/nesug10/bb/bb13.pdf

[Wicklin 2010] Rick Wicklin, "Statistical Programming with SAS/IML Software",
SAS Institute Press, 2010

[Whitlock 2010] Ian Whitlock, "A Serious Look Macro Quoting", NESUG Proceedings, 2010,
https://www.lexjansen.com/nesug/nesug10/bb/bb16.pdf

[Bewerunge 2011] Peter Bewerunge, "Calling R Functions from SAS", PHUSE Proceedings, 2011,
https://www.lexjansen.com/phuse/2011/cs/CS07.pdf

[Kahane 2011] Dalia C. Kahane, "SAS DATA Step – Compile, Execution, and the Program Data Vector",
NESUG Proceedings, 2011, https://www.lexjansen.com/nesug/nesug11/ds/ds04.pdf

[Kahane(2) 2011] Dalia C. Kahane, "SAS DATA Step Merge – A Powerful Tool", NESUG Proceedings, 2011,
https://www.lexjansen.com/nesug/nesug11/ds/ds03.pdf

[Lavery 2011] Russ Lavery, "An Animated Guide: The SAS Data Step Debugger", NESUG Proceedings, 2011,
https://www.lexjansen.com/nesug/nesug11/ds/ds10.pdf

[Loren & DeVenezia 2011] Judy Loren, Richard A. DeVenezia, "Building Provider Panels: An Application for the Hash of Hashes",
SGF Proceedings, 2011, https://support.sas.com/resources/papers/proceedings11/255-2011.pdf

[Carpenter 2012] Arthur L. Carpenter, "Carpenter's Guide to Innovative SAS Techniques",
SAS Institute Press, 2012

[Manickam 2012] Airaha Chelvakkanthan Manickam, "Interesting technical mini-bytes of Base SAS – From Data step to Macros",
SGF Proceedings, 2012, https://support.sas.com/resources/papers/proceedings12/222-2012.pdf

[Rhoads 2012] Mike Rhoads, "Use the Full Power of SAS in Your Function-Style Macros",
SGF Proceedings, 2012, https://support.sas.com/resources/papers/proceedings12/004-2012.pdf

[Secosky 2012] Jason Secosky, "Executing a PROC from a DATA Step", SGF Proceedings, 2012,
https://support.sas.com/resources/papers/proceedings12/227-2012.pdf

[Wei 2012] Xin Wei, "%PROC_R: A SAS Macro that Enables Native R Programming in the Base SAS Environment",
Journal of Statistical Software, 46(2), 1–13., 2012, https://www.jstatsoft.org/article/view/v046c02,
doi: https://doi.org/10.18637/jss.v046.c02

[Dorfman 2013] Paul M. Dorfman, "The Magnificent DO", SGF Proceedings, 2013,
https://support.sas.com/resources/papers/proceedings13/126-2013.pdf

[Henrick et al. 2013] Andrew Henrick, Donald Erdman, Stacey Christian, "Hashing in PROC FCMP to Enhance Your Productivity",
SGF Proceedings, 2013, http://support.sas.com/resources/papers/proceedings13/129-2013.pdf

[Langston 2013] Rick Langston, "Submitting SAS Code On The Side", SGF Proceedings, 2013,
https://support.sas.com/resources/papers/proceedings13/032-2013.pdf

[Lavery 2013] Russ Lavery, "Fast Access Tricks for Large Sorted SAS Files", MWSUG Proceedings, 2013,
https://www.mwsug.org/proceedings/2013/HW/MWSUG-2013-HW02.pdf

[Zender 2013] Cynthia L. Zender, "Macro Basics for New SAS Users", SGF Proceedings, 2013,
https://support.sas.com/resources/papers/proceedings13/120-2013.pdf

[Carpenter 2014] Arthur L. Carpenter, "Table Lookup Techniques: From the Basics to the Innovative",
WUSS Proceedings, 2014, https://www.lexjansen.com/wuss/2014/15_Final_Paper_PDF.pdf

[Dorfman 2014] Paul M. Dorfman, Don Henderson, "The SAS Hash Object in Action", WUSS Proceedings, 2014,
http://www.lexjansen.com/wuss/2014/113_Final_Paper_PDF.pdf

[Yang et al. 2014] Weili Yang, Fang Chen, Liping Zhang, Wenyu Hu, "Table Lookup in SAS ",
NESUG Proceedings, 2010, https://www.lexjansen.com/nesug/nesug10/cc/cc37.pdf

[Dorfman & Henderson 2015] Paul M. Dorfman, Don Henderson, "Data Aggregation Using the SAS Hash Object",
SGF Proceedings, 2015, https://support.sas.com/resources/papers/proceedings15/2000-2015.pdf

[Horstman 2015] Joshua M. Horstman, "Find your Way to Quick and Easy Table Lookups with FINDW",
MWSUG Proceedings, 2015, https://www.mwsug.org/proceedings/2015/RF/MWSUG-2015-RF-12.pdf

[Matise 2015] Joe Matise, "Unravelling the Knot of Ampersands", SGF Proceedings, 2015,
https://support.sas.com/resources/papers/proceedings15/3285-2015.pdf

[Mohammed et al. 2015] Zabiulla Mohammed, Ganesh K. Gangarajula, Pradeep Kalakota, "Working with PROC FEDSQL in SAS 9.4",
SGF Proceedings, 2015, https://support.sas.com/resources/papers/proceedings15/3390-2015.pdf

[Pillay 2015] Rahul G. Pillay, "Doing More with SAS Arrays", WUSS Proceedings, 2015,
https://www.lexjansen.com/wuss/2015/85_Final_Paper_PDF.pdf

[Tomas 2015] Paul Tomas, "Driving SAS with Lua", SGF Proceedings, 2015,
https://support.sas.com/resources/papers/proceedings15/SAS1561-2015.pdf

[Bulaienko 2016] Diana Bulaienko, "SAS and R - stop choosing, start combining and get benefits!",
PharmaSUG Proceedings, 2016, https://www.pharmasug.org/proceedings/2016/QT/PharmaSUG-2016-QT14.pdf

[Carpenter 2016] Arthur L. Carpenter, "Carpenter's Complete Guide to the SAS Macro Language, Third Edition",
SAS Institute Press, 2016

[Hu 2016] Jiangtang Hu, "New Game in Town: SAS Proc Lua with Applications",
SESUG Proceedings, 2016, https://www.lexjansen.com/sesug/2016/AD-133_Final_PDF.pdf

[Kuligowski & Mendez 2016] Andrew T. Kuligowski, Lisa Mendez, "An Introduction to SAS Arrays",
SGF Proceedings, 2016, https://support.sas.com/resources/papers/proceedings16/6406-2016.pdf

[Zdeb 2016] Mike Zdeb, "Some _FILE_ Magic", SESUG Proceedings, 2016,
https://analytics.ncsu.edu/sesug/2016/CC-171_Final_PDF.pdf

[Bayliss & Flynn 2017] Andy Bayliss, Joe Flynn, "SAS DATA Step Debugger in SAS Enterprise Guide",
PHUSE Proceedings, 2017, https://www.lexjansen.com/phuse/2017/ct/CT06.pdf

[Carpenter 2017] Arthur L. Carpenter, "Five Ways to Create Macro Variables: A Short Introduction to the Macro Language",
SGF Proceedings, 2017, https://support.sas.com/resources/papers/proceedings17/1516-2017.pdf

[Keintz 2017] Mark Keintz, "Leads and Lags: Static and Dynamic Queues in the SAS DATA STEP, 2nd ed.",
SGF Proceedings, 2017, https://support.sas.com/resources/papers/proceedings17/1277-2017.pdf

[Lafler 2017] Kirk Paul Lafler, "Advanced Programming Techniques with PROC SQL",
SGF Proceedings, 2017, https://support.sas.com/resources/papers/proceedings17/0930-2017.pdf

[Secosky 2017] Jason Secosky, "SAS DATA Step in SAS Viya: Essential New Features",
MWSUG Proceedings, 2017, https://www.lexjansen.com/mwsug/2017/BB/MWSUG-2017-BB129-SAS.pdf

[Vijayaraghavan 2017] Anand Vijayaraghavan, "A Comparison of the LUA Procedure and the SAS Macro Facility",
SGF Proceedings, 2017, https://support.sas.com/resources/papers/proceedings17/SAS0212-2017.pdf

[Carpenter 2018] Arthur L. Carpenter, "Using PROC FCMP to the Fullest: Getting Started and Doing More",
SGF Proceedings, 2018, https://www.lexjansen.com/wuss/2018/41_Final_Paper_PDF.pdf

[Carpenter(2) 2018] Arthur L. Carpenter, "Using Memory Resident Hash Tables to Manage Your Sparse Lookups",
WUSS Proceedings, 2018, https://support.sas.com/resources/papers/proceedings18/2403-2018.pdf

[Dorfman 2018] Paul M. Dorfman, "Efficient Elimination of Duplicate Data Using the MODIFY Statement",
SGF Proceedings, 2018, https://support.sas.com/resources/papers/proceedings18/2426-2018.pdf

[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

[Huffman et al. 2018] Cuyler R. Huffman, Matthew M. Lypka, Jessica L. Parker, "Anythin You Can Do I Can Do Better: PROC FEDSQL VS PROC SQL", SGF Proceedings, 2018, https://support.sas.com/resources/papers/proceedings19/3734-2019.pdf

[Jordan 2018] Mark Jordan, "Mastering the SAS DS2 Procedure: Advanced Data Wrangling Techniques, Second Edition", SAS Institute Press, 2018

[Kim 2018] Kevin Kim, "Introducing Interactive Data Step Debugger: What you can do with SAS Data Step Debugger (SAS Enterprise Guide 7.13)", PharmaSUG Proceedings, 2018, https://www.pharmasug.org/proceedings/2018/AD/PharmaSUG-2018-AD33.pdf

[McNeill et al. 2018] Bill McNeill, Andrew Henrick, Mike Whitcher, Aaron Mays, "FCMP: A Powerful SAS Procedure You Should Be Using", SGF Proceedings, 2018, https://support.sas.com/resources/papers/proceedings18/2125-2018.pdf

[Raithel 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

[Raithel(2) 2018] Michael A. Raithel, "Validating User-Submitted Data Files with Base SAS", SGF Proceedings, 2018, https://support.sas.com/resources/papers/proceedings18/1662-2018.pdf

[Renauldo 2018] Veronica Renauldo, "Efficiency Programming with Macro Variable Arrays", MWSUG Proceedings, 2018, https://www.lexjansen.com/mwsug/2018/SP/MWSUG-2018-SP-62.pdf

[Russell & Nelson 2018] Kevin Russell, Gerry Nelson, "Let's Start Something New! A Beginner's Guide to Programming in the SAS Cloud Analytic Services Environment", PharmaSUG Proceedings, 2018, https://pharmasug.org/proceedings/2018/AD/PharmaSUG-2018-AD23.pdf

[Dorfman & Shajenko 2019] Paul M. Dorfman, Lessia S. Shajenko, "Re-Mapping A Bitmap", SGF Proceedings, 2019, https://support.sas.com/resources/papers/proceedings19/3101-2019.pdf

[Horstman 2019] Joshua M. Horstman, "Using Macro Variable Lists to Create Dynamic Data-Driven Programs", MWSUG Proceedings, 2019, https://www.lexjansen.com/mwsug/2019/SP/MWSUG-2019-SP-053.pdf

[Horstman(2) 2019] Joshua M. Horstman, "Fifteen Functions to Supercharge Your SAS Code", SESUG Proceedings, 2019, https://www.lexjansen.com/sesug/2019/SESUG2019_Paper-204_Final_PDF.pdf

[Hughes 2019] Troy Martin Hughes, "User-Defined Multithreading with the SAS DS2 Procedure: Performance Testing DS2 Against Functionally Equivalent DATA Steps", PharmaSUG Proceedings, 2019, https://www.pharmasug.org/proceedings/2019/AD/PharmaSUG-2019-AD-228.pdf

[Jablonski 2019] Bartosz Jabłoński, "Use the Advantage of INDEXes Even If a WHERE Clause Contains an OR Condition", SGF Proceedings, 2019, https://support.sas.com/resources/papers/proceedings19/3722-2019.pdf

[Khorlo 2019] Igor Khorlo, "Automating Migration from the SAS Macro Language to the LUA Procedure Using Transpiling", SGF Proceedings, 2019, https://support.sas.com/resources/papers/proceedings19/3824-2019.pdf

[Lafler 2019] Kirk Paul Lafler, "PROC SQL: Beyond the Basics Using SAS, Third Edition", SAS Institute Press, 2019

[Morioka 2019] Yutaka Morioka, "DOSUBL Function + SQL View + Hash Object FedSQL + PROC DS2 Hash Package", SGF Proceedings, 2019, https://support.sas.com/resources/papers/proceedings19/3128-2019.pdf

[Mukherjee 2019] Chad Mukherjee, "FETCH()ing Use Cases for the Data Access Functions", SGF Proceedings, 2019, https://support.sas.com/resources/papers/proceedings19/3607-2019.pdf

[Iyengar & Horstman 2020] Jay Iyengar, Josh Horstman, "Look Up Not Down: Advanced Table Lookups in Base SAS", SESUG Proceedings, 2020, https://www.lexjansen.com/sesug/2020/SESUG2020_Paper_150_Final_PDF.pdf

[Jablonski 2020] Bartosz Jabłoński, "SAS Packages: The Way to Share (a How To)", SGF Proceedings, 2020, 4725-2020
https://www.sas.com/content/dam/SAS/support/en/sas-global-forum-proceedings/2020/4725-2020.pdf
extended version available at: https://github.com/yabwon/SAS_PACKAGES/blob/main/SPF/Documentation

[McMullen 2020] Quentin McMullen, "A Close Look at How DOSUBL Handles Macro Variable Scope", SGF Proceedings, 2020, https://support.sas.com/resources/papers/proceedings20/4958-2020.pdf

[Sober & Kinnebrew 2020] Steven Sober, Brian Kinnebrew, "Best Practices for Converting SAS Code to Leverage SAS Cloud Analytic Services", SGF Proceedings, 2020, https://support.sas.com/resources/papers/proceedings20/4147-2020.pdf
GitHub: https://github.com/sascommunities/sas-global-forum-2020/tree/master/papers/4147-2020-Sober

[Jablonski 2021] Bartosz Jabłoński, "My First SAS Package - a How To", SGF Proceedings, 2021 , 1079-2021
https://communities.sas.com/kntur85557/attachments/kntur85557/proceedings-2021/59/1/Paper_1079-2021.pdf
also available at: https://github.com/yabwon/SAS_PACKAGES/tree/main/SPF/Documentation/Paper_1079-2021

[Watson & Hadden 2021] Richann Watson, Louise Hadden, "What Kind of WHICH Do You CHOOSE to be?", SESUG Proceedings, 2021, https://www.lexjansen.com/sesug/2021/SESUG2021_Paper_37_Final_PDF.pdf

[Bremser 2022] Kurt Bremser, "Talking to Your Host Interacting with the Operating System and File System from SAS", WUSS Proceedings, 2022, https://www.lexjansen.com/wuss/2022/WUSS-2022-Paper-24.pdf

[Box 2023] Jim Box, "Running Python Code Inside a SAS Program", PharmaSUG Proceedings, 2023, https://www.pharmasug.org/proceedings/2023/QT/PharmaSUG-2023-QT-165.pdf

[Gilsen 2023] Bruce Gilsen, "Using the R interface in SAS to Call R Functions and Transfer Data", PharmaSUG Proceedings, 2023, https://www.pharmasug.org/proceedings/2023/AP/PharmaSUG-2023-AP-079.pdf

[Jablonski 2023] Bartosz Jabłoński, "Share your code with SAS Packages a Hands-on-Workshop",
WUSS 2023 Proceedings, 208-2023, https://www.lexjansen.com/wuss/2023/WUSS-2023-Paper-208.pdf

[Jablonski(2) 2023] Bartosz Jabłoński, "A SAS Code Hidden in Plain Sight",
WUSS 2023 Proceedings, 208-2023, https://www.lexjansen.com/wuss/2023/WUSS-2023-Paper-189.pdf

[Lankham & Slaughter 2023] Isaiah Lankham, Matthew T. Slaughter, "Friends are better with Everything: A User's Guide to PROC FCMP Python Objects in Base SAS",
PharmaSUG Proceedings, 2023, https://www.lexjansen.com/pharmasug/2023/AP/PharmaSUG-2023-AP-049.pdf

[Hughes 2024] Troy Martin Hughes, "PROC FCMP User-Defined Functions: An Introduction to the SAS Function Compiler",
SAS Institute Press, 2024

[Jablonski 2024] Bartosz Jabłoński, "Macro Variable Arrays Made Easy with macroArray SAS Package",
PharmaSUG Proceedings, 2024, https://www.lexjansen.com/pharmasug/2024/AP/PharmaSUG-2024-AP-108.pdf

[Jablonski & McMullen 2024] Bartosz Jabłoński, Quentin McMullen, "Fifty Shades of SAS Programming, or 50 (+3) Syntax Snippets for a Table Look-up Task, or How to Learn SAS by Solving Only One Exercise!",
WUSS Proceedings, 2024, https://www.lexjansen.com/wuss/2024/125_FINAL_paper_pdf.pdf

RECORDINGS

[Barr 2018(video)] Anthony James Barr, "From Sir Ronald Fisher to SAS76", The 14th SUGUKI meetup, 2018, (start at 29:04)
Recording: https://www.youtube.com/watch?v=-qa_vufvj5g
Event details: https://www.meetup.com/suguki/events/247371376/

[Jablonski 2023(video)] Bartosz Jabłoński, "SAS Packages Framework - an easy code sharing medium for SAS", Warsaw IT Days, 2023
Recording: https://youtu.be/T52Omisi0dk&t=0s

ACKNOWLEDGMENTS

We would like to thank Norbert Trojan for his contribution to idea in look-up 10 (the `set` statement with multiple data sets).
We would like to thank Philip Holland, Ron Fehd, Erik Eknor Ackzell, and Troy Martin Hughes for their invaluable contribution in "polishing" this text.

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.
Contact Quentin at e-mail address:

qmcmullen✉gmail.com

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

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.
```

Appendix B - install the SPF and packages

To install the SAS Packages Framework and a SAS Package we execute the following steps:

- First we create a directory to install SPF and Packages, for example: `/home/user/packages` or `C:/packages`.
- Next, depending if the SAS session has access to the internet:
  - if it does - we run the following code:

code: install from the internet

```
filename packages "/home/user/packages";
filename SPFinit url
"https://raw.githubusercontent.com/yabwon/SAS_PACKAGES/main/SPF/SPFinit.sas";
%include SPFinit;

%installPackage(SPFinit)
%installPackage(packageNameYouWant)
```

  - If the SAS session does not have access to the internet we go to the framework repository:

    https://github.com/yabwon/SAS_PACKAGES

    next (if not already) we click the stargazer button [★] ;-) and then we navigate to the `SPF` directory and we copy the `SPFinit.sas` file into the directory from step one (direct link: `https://raw.githubusercontent.com/yabwon/SAS_PACKAGES/main/SPF/SPFinit.sas`). And for packages - we just copy the package zip file into the directory from step one.
- From now on, in all subsequent SAS session, it is enough to just run:

code: enable framework and load packages

```
filename packages "/home/user/packages";
%include packages(SPFinit.sas);
%loadPackage(packageNameYouWant)
```

to enable the framework and load packages. To update the framework or a package to the latest version we simply run:

code: update from the internet

```
%installPackage(SPFinit packageName1 packageName2 packageName3)
```

Appendix C - safety considerations

The SPF installation process, in a "nutshell", reduces to copying the `SPFinit.sas` file into the packages directory. It is the same for a packages too.

You may ask: *is it safe to install*?

Yes, it's safe! When you install the SAS Packages Framework, and later when you install packages, the files are simply copied into the packages directory that you configured above. There are no changes made to your SAS configuration files, or autoexec, or registry, or anything else that could somehow "break SAS." As you saw, you can perform a manual installation simply by copying the files yourself. Furthermore the SAS Packages Framework is:

- written in 100% SAS code, it does not require any additional external software to work,
- full open source (and MIT licensed), so every macro can be inspected.

When we work with a package, before we even start thinking about loading content of one into the SAS session, both the help information and the source code preview are available.

To *read help information* (printed in the log) you simply run:

code: get help info

```
%helpPackage(<packageName>, <*|componentName|license>)
```

To **preview source code of package components** (also printed in the log) you simply run:

code: get code preview

```
%previewPackage(<packageName>, <*|componentName>)
```

The asterisk means "print everything", the `componentName` is the name of a macro, or a function, or a format, etc. you want see.

Appendix D - "sevenfold" indirect referencing

code: sevenfold indirect referencing

```
%let VeryImportantMessage=This, is, PharmaSUG!!!;

%let T1=Very;
%let T2=Important;
%let T3=Message;

%let letter=T;
%let one=1;
%let two=2;
%let three=3;

%put &&&&&&&letter&one&&&letter&two&&&letter&three;
```

How the resolution goes? The `{}` are added to indicate grouping.

code: first grouping and resolving

```
{&&}{&&}{&&}{&letter}{&one}{&&}{&letter}{&two}{&&}{&letter}{&three}
```

the log - first intermediate result

```
{&}{&}{&}{T}{1}{&}{T}{2}{&}{T}{3}
```

code: second grouping and resolving

```
1 {&&}{&T1}{&T2}{&T3}
```

the log - second intermediate result

```
1 {&}{Very}{Important}{Message}
```

code: third grouping and resolving

```
1 {&VeryImportantMessage}
```

the log - result

```
1 This, is, PharmaSUG!!!
```

Appendix E - SAS releases history

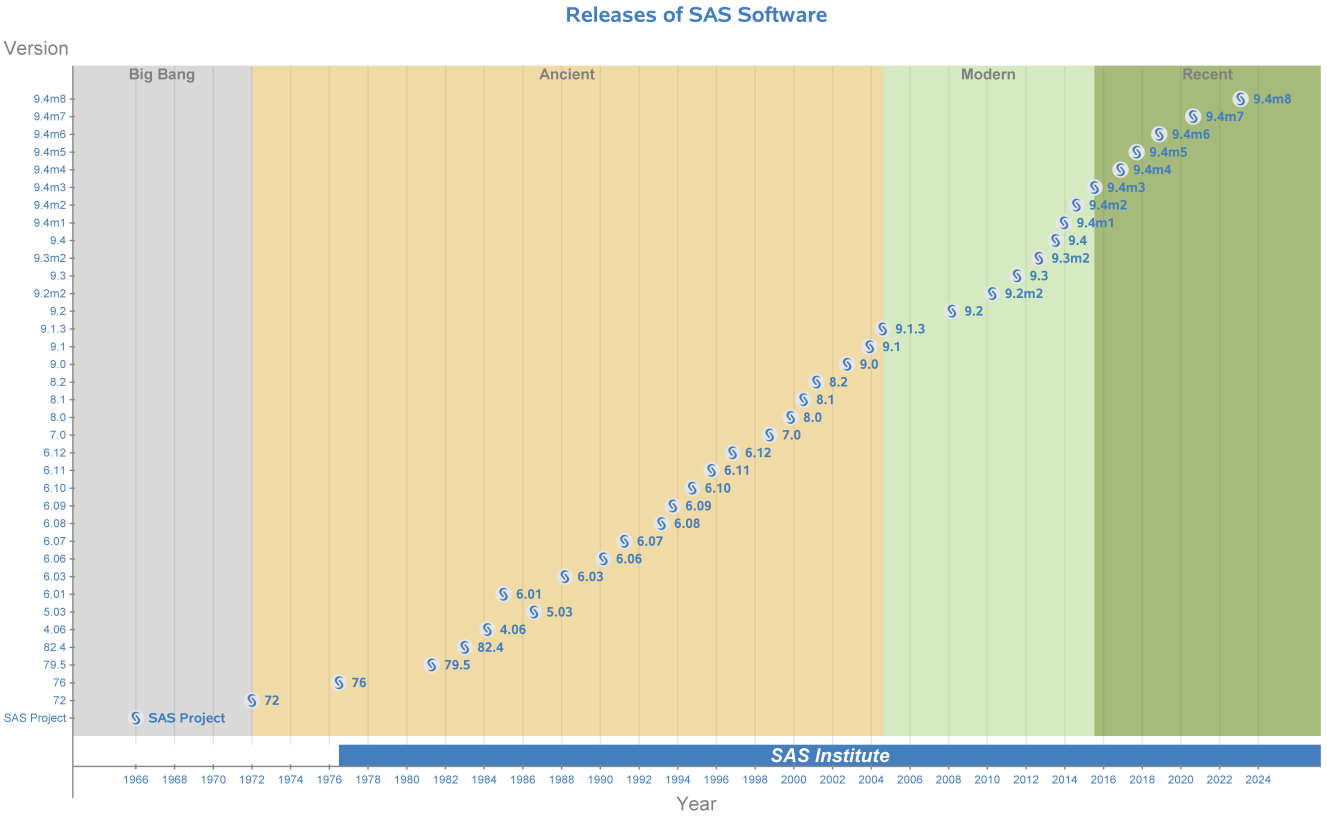

Figure 1: SAS releases history

INDEX