Look Up Not Down: Advanced Table Lookups in Base SAS

Jay Iyengar, Data Systems Consultants, Oak Brook, IL
Josh Horstman, Nested Loop Consulting, Indianapolis, IN

ABSTRACT

One of the most common data manipulation tasks SAS programmers perform is combining tables through table lookups. In the SAS programmer's toolkit many constructs are available for performing table lookups. Traditional methods for performing table lookups include conditional logic, match-merging and SQL joins. In this paper we concentrate on advanced table lookup methods such as formats, multiple SET statements, and HASH objects. We conceptually examine what advantages they provide the SAS programmer over basic methods. We also discuss and assess performance and efficiency considerations through practical examples.

INTRODUCTION

Within Base SAS, the two most commonly used table lookup techniques are the DATA step merge, and the Proc Sql join. Most SAS users are quite familiar with these techniques and employ them extensively. In our experience, the DATA step merge is considered to be the more complex method, where the Proc Sql join is considered to be more intuitive and user-friendly. For this paper, we define these techniques as traditional table lookup methods. Opinions vary on the performance and efficiency of these two techniques, for given programming situations. Because of the attention they receive in the SAS literature, and their exclusion from many SAS programming courses, many SAS users aren't aware of advanced techniques for combining input tables or SAS data sets. However, SAS contains advanced capabilities for performing table lookups. More experienced SAS users, especially Certified SAS Programmers are more familiar with table lookup methods.

WHAT ARE TABLE LOOKUPS?

With the advent of databases, a process was devised for organizing data structures in database objects. Part of this process involved deciding how to collect and store large amounts of data within a database. Distinct pieces of information would be stored in separate columns, called fields in database terminology. As databases grew in scale, it became practical and efficient to store data in separate database objects. Databases containing large numbers of variables could store sets of variables in unique tables. This was the idea of the relational database. Specific variables could be retrieved from their respective tables by linking to their table. This is the basic concept of a table lookup.

In terms of the structure of a table lookup, a table lookup is performed when two or more data sets are combined horizontally. Table is just an informal name for a SAS data set. The tables are joined based on the value of a key variable or primary key. Each table is indexed on a primary key, which is common to both data sets. Technically, the primary key should be a way to uniquely identify records in a dataset. For example, many healthcare data sets use Patient ID as a primary key. In a DATA step merge, the primary key is the BY variable, specified in a BY statement. In a PROC SQL join, the primary key is the variable specified in a join condition in the WHERE or ON clause.

Amongst SAS users, table lookups have been used interchangeably with merges and joins. In the simple example of a table lookup, you have a base table which contains the bulk of your data, and a lookup table which contains only a few variables. The base table can be thought of as a master data set containing millions of records or observations, where the lookup table is usually much smaller. This example of a table lookup is directly relevant to healthcare data sets which can be very massive. Healthcare data sets may contain millions of patient records, and include codes to indicate medical diagnoses or procedures. They need to be linked to smaller reference tables to obtain the definitions of those codes. Figure 1 displays a schematic of a lookup involving five tables.
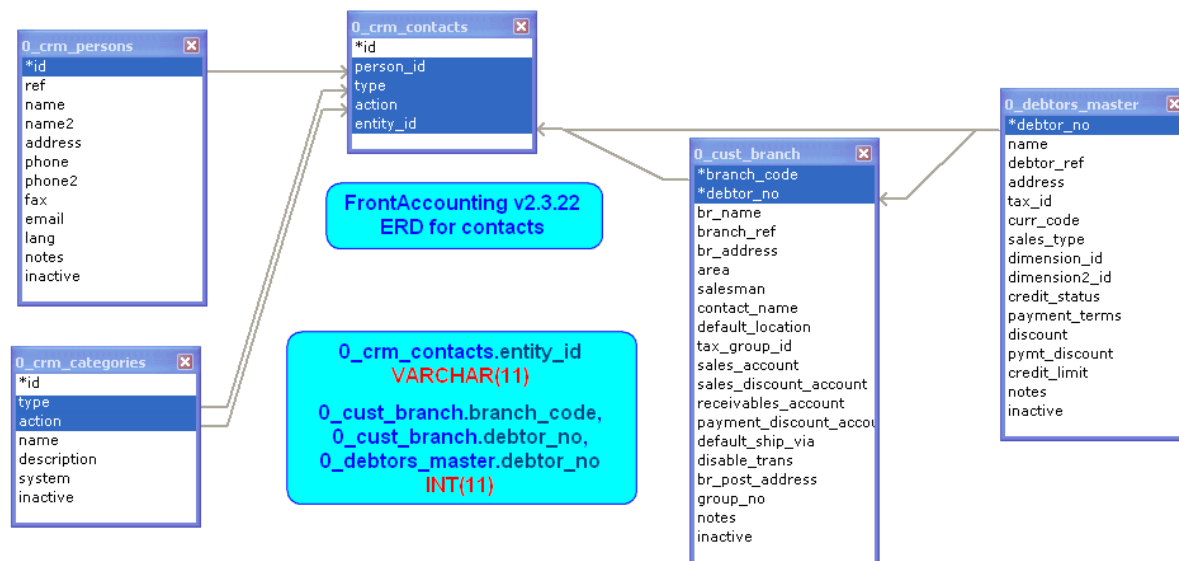
Figure 1. Table Lookup Schema

METHODS FOR TABLE LOOKUPS

There are many methods available for performing table lookups using Base SAS software, ranging from the simple to the complex. Carpenter (2001 and 2015) details several such methods.

Selecting a method to use for a given situation may depend on a number of factors. When dealing with large datasets, efficient performance may be a priority. In other cases, it may be desirable to keep the code as simple as possible for ease of maintenance and reuse. Each table lookup method has its own advantages and disadvantages, which must be considered in the context of the task being performed.

It is advantageous for the SAS programmer to be familiar with a variety of methods. Although this paper focuses on several more advanced techniques, we begin with a review of several simple approaches commonly used.

SIMPLE TABLE LOOKUP METHODS

At its core, a table lookup is simply retrieving a value for one variable based on the value for another. One of the simplest methods for accomplishing this is by hard-coding the relationship between the two variables directly into the programming logic. This is often done through a series of IF…THEN…ELSE statements or a SELECT statement. Horstman (2017) presents a method for hard-coding a table lookup using the WHICHC/WHICHN and CHOOSEC/CHOOSEN DATA step functions. These approaches are quick and easy when the lookup values are few in number and unlikely to change, but they do not scale well.

Another way of performing a table lookup is by combining multiple data sets using the MERGE statement in the DATA step. This is a very common approach because it is simple and versatile. However, it does have some limitations. One notable drawback is that data sets must generally be sorted before they can be merged in this manner. When data sets are large, sorting can be a costly operation. In addition, there are other pitfalls associated with the DATA step merge. Horstman (2018) details several potential problems.

SQL joins are another common means of accomplishing a table lookup. One advantage of a SQL join over a DATA step merge is that there is typically no need to sort the incoming data sets prior to a SQL join. Thus, the table lookup can be completed with a minimal amount of code. However, SQL joins can be quite resource-intensive in certain situations.

ADVANCED TABLE LOOKUP METHODS

Although the simple table lookup methods listed above are sufficient for the majority of programming tasks, there are situations when a more advanced procedure can be advantageous. This paper examines the following five methods:

2

1. Formats
2. hash tables
3. Multiple SET Statements
4. Indexes with the KEY= Option
5. Arrays and the DOW Loop

METHOD #1: FORMATS

One of the traditional ways to perform a table lookup is through the use of a SAS format. Although this technique is relatively simple, we've chosen to explore formats because they present an alternative to the DATA step merge and PROC SQL join and are used less frequently. From that standpoint, and for the purposes of this paper, formats can be considered an advanced lookup technique.

To perform a table lookup using formats, the SAS user creates a user-defined format using PROC FORMAT. The key values and the data values of a user-defined format can be coded explicitly in a PROC FORMAT VALUE statement or based on variable values in a SAS data set. Thus, the lookup table may not be a table at all depending on the method chosen by the programmer. If the programmer creates the format from a SAS data set, they must prepare a SAS data set having a specific structure that contains the key values and data values. Alternatively, the lookup values can be embedded directly in the code and hard-coded and mapped to key values in the PROC FORMAT step.

PROC FORMAT WITH PUT FUNCTION

```
                    /* Map Race Code to Racial Description */
          Proc Format;
               Value $RaceEth
               1 = 'White'
               2 = 'Black'
               3 = 'Hispanic'
               5 = 'Asian Pacific Islander';
 NOTE: Format $RACEETH has been output.
          Run;
 NOTE: PROCEDURE FORMAT used (Total process time):
       real time            0.06 seconds
       cpu time             0.03 seconds


                    /* Derive New Variable containing Race Description */
          Data CARRIER_CLAIM_2010;
            Set SASDATA1.CARRIER_CLAIM_2010;
             RaceEth_Description = Put(Race, $RaceEth.);
          Run;

 NOTE: There were 1163957 observations read from the data set SASDATA1.CARRIER_CLAIM_2010.
 NOTE: The data set WORK.CARRIER_CLAIM_2010 has 1163957 observations and 14 variables.
 NOTE: DATA statement used (Total process time):
       real time            3.86 seconds
       cpu time             1.54 seconds
```

Figure 2. Proc Format and DATA step using Put Function.

In the example in Figure 2, a user-defined format, RACEETH, is created in the first step using PROC FORMAT. The RACEETH format assigns descriptive labels for racial categories to numeric values. The format is defined as numeric because the key values are numeric, although the values being assigned are character data. The format specifies a

limited, small number of values and labels. This is typical for demographic variables which are commonly discrete or nominal, as opposed to continuous.

In the second step, a DATA step reads the SAS data set CARRIER_CLAIM_2010, which contains Medicare claims data for 2010. CARRIER_CLAIM_2010 is the base table. As indicated in the SAS log, it's a large data set, containing 1,163,957 records. RACE is a variable in CARRIER_CLAIM_2010 which contains numeric codes. As we stated earlier, administrative healthcare datasets contain codes to indicate diagnoses, procedures, physicians, etc.

In this case, the format RACEETH contains the lookup values, instead of them being stored in a separate reference table. The lookup is performed by applying the newly created format to the RACE variable and storing the result in a new variable. An assignment statement and a PUT function are used to create the RACEETH_DESCRIPTION variable. The format RACEETH is referenced in the PUT function and applied to the variable RACE.

One drawback of this technique is that it can require a lot of maintenance. A demographic variable such as race is well-suited to this technique because it contains a small set of distinct values that don't change frequently. If the set of values were in the hundreds, imagine how tedious and arduous a task it would be for the programmer to hard-code all those unique values. Also, suppose the values need to be updated or values need to be added to the format. In this case, the programmer must manually edit the hard-coded values. This introduces the risk of typos or mistakes being made which might compromise the validity of the data.

In this example, applying the format with a PUT function executed efficiently. In terms of performance, the table lookup processed in a very short amount of time with a base table of over one million records. The log in Figure 1 shows the execution took less than 4 seconds of real time, and under 2 seconds of CPU. However, the technique has the disadvantage of maintainability, because it requires manual editing. Similarly, it isn't practical with a large volume of lookup values. A better solution would be to utilize a data-driven method.

PROC FORMAT CNTLIN SAS DATA SET

In our previous example, we concluded the PROC FORMAT and PUT function method was inappropriate for table lookups where the unique lookup values were in the hundreds or thousands. For example, administrative healthcare data sets contain ICD9 or ICD10 diagnosis codes. There can be as many as 15,000 to 17,000 individual ICD9 or ICD10 diagnosis codes. Using the CNTLIN= option in PROC FORMAT, SAS has versatile capabilities for creating formats to cope with such a large volume of data values. With this construct, SAS can perform effective table lookups where the lookup values are on a large scale.

SAS can create a format from a SAS data set which contains the lookup values. In PROC FORMAT, you need to specify the SAS data set using the CNTLIN= option. There are several requirements for a CNTLIN data set. It needs to have several required variables. START contains the values in a VALUE statement from the left side of the equals sign. LABEL contains the values from the right-side of the equal sign or the lookup values.

In Figure 2 below, the excerpt from the SAS log illustrates a DATA step which manipulates the SAS data set, ICD9DX. The data set ICD9DX contains over 17,300 unique medical diagnosis codes. The required variables for a CNTLIN data set are START, LABEL, and FMTNAME. The DATA step prepares ICD9DX to meet the requirements of a CNTLIN data set. TYPE, which is assigned 'C', is an optional variable which indicates the type of format, character or numeric. For a format based on a range of numeric values, an additional variable, END, would be needed.

```
                     /* Preparing the CNTLIN SAS Data Set */
        Data ICD9DX;
           Set advtbllu.icd9dx(Rename=(Description=Label));

           Retain Fmtname '$ICDDX' Type 'C';

           Start = Compress(code, '.');

           Keep Type Start Label Fmtname;
        Run;

  NOTE: There were 17374 observations read from the data set ADVTBLLU.ICD9DX.
  NOTE: The data set WORK.ICD9DX has 17374 observations and 4 variables.
  NOTE: DATA statement used (Total process time):
        real time            0.03 seconds
        cpu time             0.02 seconds


       Proc Format Cntlin=ICD9DX;
NOTE: Format $ICDDX has been output.
       Run;

NOTE: PROCEDURE FORMAT used (Total process time):
        real time            0.19 seconds
        cpu time             0.18 seconds

NOTE: There were 17374 observations read from the data set WORK.ICD9DX.

      Data carrier_claim_2010;
           Set advtbllu.carrier_claim_2010;
           DX1 = Put(ICD9_DGNS_CD_1, $ICDDX.);
        Run;

NOTE: There were 1163957 observations read from the data set ADVTBLLU.CARRIER_CLAIM_2010.
NOTE: The data set WORK.CARRIER_CLAIM_2010 has 1163957 observations and 10 variables.

NOTE: DATA statement used (Total process time):
        real time            2.97 seconds
        cpu time             2.64 seconds
```

Figure 3. DATA step to prepare CNTLIN data set and PROC FORMAT with CNTLIN= option.

In the SAS log in Figure 3 above, SAS creates and outputs the format $ICDDX based on the data set ICD9DX. The data set ICD9DX contains both the key and lookup values which are stored in $ICDDX once it's created. The coding for the PROC FORMAT CNTLIN is very simple. Once again, the table lookup is executed by applying the format to a variable, ICD9_DGNS_CD_1, on the CARRIER_CLAIM_2010 using the PUT function, and storing the result in the newly created variable, DX1, using a simple assignment statement.

In performance, PROC FORMAT CNTLIN matched if not exceeded the efficiency of the previous technique. As shown in Figure 3, the execution took less than 3 seconds of real time, and under 3 seconds of CPU. Our example demonstrates the suitability of this method to handle large lookup tables. Also, the technique has the advantage of providing a data driven solution to a table lookup. If changes need to happen to the data, they can happen at the data set level without lots of manual intervention by the programmer at the code level.

METHOD #2: HASH TABLES

The hash object has been available in the Base SAS package since version 9. HASH tables are lookup tables defined using hash objects. Hash objects are commonly referred to as hash tables. Technically, the hash object is a DATA step component object consisting of attributes and methods. The hash object must be declared and instantiated within a DATA step. Once defined, hash objects are stored in memory for the duration of the DATA step. During processing data values are accessed and retrieved from the table in memory. Thus, hash tables use in-memory processing as opposed to disk-based processing.

The tabular structure of a hash object is similar to a SAS data set and contains rows and columns. The hash table contains a key component and a data component, analogous to key values and data values respectively. The key values must be unique and serve as an index or primary key for mapping to the data values. Key and data variables can be single, or composite, consisting of multiple variables, and allow storage of multiple key and data values. The hash table has the versatile feature of allowing key and data variables to be character, numeric or a mixture of the two. Data loaded into a hash object can be hard-coded in a DATA step, or loaded directly from a SAS data set. Unlike a DATA step merge, a hash table lookup doesn't require data to be sorted or indexed prior to processing.

HASH TABLE CODING

In the **DECLARE HASH** statement, the hash object is invoked, the hash table is given a name, and dimensions for the hash table are defined. If the hash table is loaded from a SAS data set, it's specified in parentheses and double quotes as shown below. If this is the case, it's not necessary to explicitly define the hash table dimensions in terms of rows and columns. Otherwise, the number of buckets or total cells for the hash table needs to be specified in parentheses.

> **Declare Hash** *'HashTableName' ("libref.sasdatasetname");*

The next sets of statements initialize the variables that the declared hash table will contain. The **DefineKey** method defines the key variables of the hash table, which serves as a primary key, and indexes the hash table. The key variables are specified in parentheses and double-quotes. The key variables can be taken from the SAS data set upon which the hash table is based.

> *HashTableName.***DefineKey ("Key Variable(s)");**

The **DefineData** property method defines the data component of the hash table. The data variable contains the actual lookup values in which we're interested. The data variable is specified in parentheses and double-quotes and can be taken from the SAS data set on which the hash table is based. There can be multiple data variables defined within the hash table.

> *HashTableName.***DefineData ("Data Variable(s)");**

The **DefineDone** property method completes the declaration of the hash table. It indicates that the dimensions, SAS data set, and variables have all been specified. The End statement concludes the block of code which defines the hash table.

> *HashTableName.***DefineDone;**
> **End;**

DATA STEP PROCESSING USING HASH TABLES

It is important to understand how a HASH object processes and performs a table lookup. To understand this, it's necessary to look behind the scenes at what SAS is doing when it executes code. One of the internal constructs which the DATA step uses is the Program Data Vector (PDV). The PDV holds one record of data at a time as it's processed by DATA step code during each iteration or loop of the DATA step. The following diagram shows what happens to the PDV as each line of DATA step code is executed in a hash table lookup operation.

```
Data BENE_MATCH BENE_NO_MATCH;
    If 0 Then Set sasfiles.finder_attrib;
```
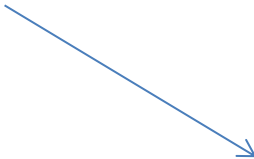
			PDV			
BENE_ID	PRFNPI	EOF1	CLM_ID	RC	_ERROR_	_N_
		0		.	0	1

Figure 4. Set Statement and PDV

The first line of code is a non-executing SET statement. When you specify "IF 0 Then Set", SAS compiles the SET statement, and adds the variables contained in the data set FINDER_ATTRIB to the PDV. If it was an executing SET statement, the PDV would be populated with the first record of data from FINDER_ATTRIB. Instead SAS initializes variables to missing. The variables from the other SAS data set have been added to the PDV as well. In addition the PDV contains any temporary variables defined during the step (EOF1), and the automatic variables _ERROR_ and _N_, the latter of which counts DATA step iterations.

```
If _N_=1 Then Do;

Declare Hash Provider (dataset: 'sasfiles.finder_attrib');
    Provider.DefineKey ('BENE_ID');
    Provider.Definedata('PrfNPI');
    Provider.Definedone ();
End;
```

			PDV			
BENE_ID	PRFNPI	EOF1	CLM_ID	RC	_ERROR_	_N_
		0		.	0	1

Figure 5. Hash Object Creation and PDV.

In the next step, SAS initializes and declares the hash object only in the first iteration of the DATA step (IF _N_=1). The hash object is created in memory. The hash table, PROVIDER, is loaded with data from the data set SASFILES.FINDER_ATTRIB. Key and data variables are defined for the PROVIDER hash table. No changes occur in the PDV because the hash table resides in memory and doesn't require any data processing to take place. Nothing from the hash object is added to the PDV.

```
Do Until (EOF1);
        Set sasfiles.carrier2010 end=EOF1;
```

PDV

BENE_ID	PRFNPI	EOF1	CLM_ID	RC	_ERROR_	_N_
00013D2EFD8E45D1		0	887243388666441	.	0	1

Figure 6. Second SET statement executes

Inside the DO Loop, SAS executes the SET statement and reads the data set SASFILES.CARRIER2010. The temporary variable EOF1 is defined using the END= option. The first observation in SASFILES.CARRIER2010 is read and loaded into the PDV. The data set SASFILES.CARRIER2010 actually contains many more variables than can be shown here. The full data set is listed in the appendix.

```
Call Missing (PRFNPI);
```

PDV

BENE_ID	PRFNPI	EOF1	CLM_ID	RC	_ERROR_	_N_
00013D2EFD8E45D1		0	887243388666441	.	0	1

Figure 7. CALL MISSING

The CALL MISSING routine initializes the values of the lookup variable PRFNPI to missing.

```
Do Until (EOF1);
        Set sasfiles.carrier2010 end=EOF1;
        Call Missing (PRFNPI);
        RC = Provider.find();
```

PDV

BENE_ID	PRFNPI	EOF1	CLM_ID	RC	_ERROR_	_N_
00013D2EFD8E45D1	2657039888	0	887243388666441	0	0	1

Figure 8. The Find Method

Using the Find Method, SAS searches the hash table PROVIDER using BENE_ID as the key variable, and the value of BENE_ID from the first record of SAS data set CARRIER2010 that's in the PDV. A match in the hash table PROVIDER is found, and SAS populates PRFNPI in the PDV from the value in the hash table. The variable RC (Return Code) is created and indicates the result of the FIND method. It's assigned RC=0 because the FIND method successfully found a match.

```
Do Until (EOF1);
        Set sasfiles.carrier2010 end=EOF1;
        Call Missing (PRFNPI);
        RC = Drgcode.find();
        If RC=0 Then Output BENE_MATCH;
```

PDV						
BENE_ID	PRFNPI	EOF1	CLM_ID	RC	_ERROR_	_N_
00013D2EFD8E45D1	2657039888	0	887243388666441	0	0	1

Figure 9. Output to SAS data set.

Because the FIND method was successful (RC=0), an IF-THEN statement writes the record in the PDV to the output data set, BENE_MATCH. This ends the first iteration of the DATA step.

HASH TABLE EXAMPLES

The Data Set

The data sets used for this hash table example comes from the GDELT (Global Database of Events, Language, and Tone) Project. The GDELT Project built a worldwide database of international conflicts and interactions between countries around the world. The database stores records of events covered by different media networks and news outlets. It was compiled from hundreds of print, broadcast, and online news sources. The full database consists of 250 million historical records dating back from the present day to 1979 and is updated periodically. For the examples using hash tables, a subset of the full database was used. This subset was used as the base table in our hash table example. A PROC CONTENTS of the GDELT data set is provided in the appendix. A snapshot of the GDELT data set is displayed in Figure 10 below.

GlobalEventID	Day	Actor1Code	Actor1Name	Actor2Code	Actor2Name	EventCode	AvgTone
11409131	19930101	CHNGOV	LI PENG	CHN	CHINA	20	8.759124088
11409132	19930101	CHNGOV	JIANG ZEMIN	CHN	CHINESE	40	15.18386714
11409133	19930101	CHNGOV	LI PENG	CHN	CHINA	40	8.759124088
11409134	19930101	CHNGOV	JIANG ZEMIN	CHN	CHINA	43	9.230769231
11409135	19930101	CHNGOV	JIANG ZEMIN	CHN	CHINA	43	9.230769231
11409136	19930101	CHNGOV	LI PENG	CHN	CHINA	51	8.759124088
11409137	19930101	CHNGOV	CHINA	CHN	CHINA	100	13.22645291

Figure 10. GDELT data set

The lookup table for this hash table example is much smaller and contains event descriptions for the event codes in the base table. A sample of the lookup table is provided in Figure 11 below.

CAMEOEVENTCODE	EVENTDESCRIPTION
1	MAKE PUBLIC STATEMENT
10	Make statement, not specified below
11	Decline comment
12	Make pessimistic comment
13	Make optimistic comment
14	Consider policy option
15h	Acknowledge or claim responsibility

Figure 11. Event Code Lookup Table

A series of tests were run on the data sets using the hash table LOOKUP method. The tests were performed using SAS University Edition version 3.8. SAS University Edition is run on a virtual machine that includes 3.5 GB of random-access memory (RAM) and 11.5 GB of storage space. SAS University Edition is known to freeze or crash when the number of observations exceeds 10 million. This is due to storage space or space allocation limitations for output data sets. As a result, the amount of data read from the base table was limited to 5 million observations.

```
        Data CODE_MATCH CODE_No_Match;
          If 0 Then Set cameo_event_codes;

          IF _N_=1 Then Do;
                Declare Hash CODE (dataset: 'cameo_event_codes');
                CODE.DefineKey ('EVENTCODE');
                CODE.DefineData ('EVENTDESCRIPTION');
                CODE.DefineDone();
           End;

           Do Until (EOF1);
                Set advtbllu.gdelt_all(obs=5000000) End=EOF1;
                Call Missing (EVENTDESCRIPTION);
                RC = CODE.Find();
                IF RC=0 Then Output CODE_MATCH;
                Else IF RC^=0 Then Output CODE_No_Match;
            End;

        Run;

 NOTE: There were 299 observations read from the data set WORK.CAMEO_EVENT_CODES.
 NOTE: There were 5000000 observations read from the data set ADVTBLLU.GDELT_ALL.
 NOTE: The data set WORK.CODE_MATCH has 4998980 observations and 11 variables.
 NOTE: The data set WORK.CODE_NO_MATCH has 1020 observations and 11 variables.
 NOTE: DATA statement used (Total process time):
       real time           14.92 seconds
       cpu time            7.95 seconds
```

Figure 12. hash table-Table Lookup

The hash table lookup example in Figure 12 illustrates a many-to-one merge. The base table contains duplicate records for distinct values of the key variable EventCode. The lookup table contains a single record for each unique value of EventCode. The OBS= option was used to limit the number of observations read and processed from the Base Table to 5,000,000. The LOOKUP table of event codes contains 300 observations. The output of the table

lookup includes separate data sets for matches and non-matches based on the return code. As indicated in the SAS log in Figure 12, the execution of the table lookup completed in less than 15 seconds of real time.

In order to assess the relative efficiency of the hash table technique, this exercise was repeated using conventional table lookup techniques. The example was repeated using the common DATA step merge and PROC SQL join constructs.

```
                           /* Data Step Merge Example */
 Proc Sort Data=advtbllu.gdelt_all(Obs=5000000) Out=GDELT_ALL;
     By EVENTCODE;
 Run;

 NOTE: There were 5000000 observations read from the data set ADVTBLLU.GDELT_ALL.
 NOTE: PROCEDURE SORT used (Total process time):
       real time            21.88 seconds
       cpu time             18.05 seconds

 Proc Sort Data = advtbllu.cameo_event_codes(Rename=(cameoeventcode=eventcode))
              out=cameo_event_codes Nodupkey;
     By EVENTCODE;
 Run;

 NOTE: The data set WORK.CAMEO_EVENT_CODES has 299 observations and 2 variables.
 NOTE: PROCEDURE SORT used (Total process time):
       real time            0.63 seconds
       cpu time             0.03 seconds

 Data AdmHlth.gdelt_all_v2;
     Merge gdelt_all (IN=A) cameo_event_codes (IN=B);
        By EVENTCODE;
     If A and B;
 Run;

 NOTE: There were 5000000 observations read from the data set WORK.GDELT_ALL.
 NOTE: There were 299 observations read from the data set WORK.CAMEO_EVENT_CODES.
 NOTE: The data set ADMHLTH.GDELT_ALL_V2 has 4998980 observations and 10 variables.
 NOTE: DATA statement used (Total process time):
       real time            30.24 seconds
       cpu time             6.42 seconds
```

Figure 13. DATA step Merge Table Lookup

In Figure 13 above is the SAS log from the retest of the same example using the DATA step merge. Repeating the example with the DATA step merge resulted in a total execution time of 52.75 seconds of real time, and 24.75 seconds of CPU, as displayed in Figure 13. This is three times the amount of CPU time and over three times the amount of real time it took to execute the table lookup using hash tables. CPU time is appropriate as a metric of efficiency here because both tests were performed using the same platform and system.

Figure 14 displays the SAS log from a retest using a PROC SQL join. A third run of the same example using a PROC SQL inner join resulted in an execution of 35.58 seconds of real time, and only 8.14 seconds of CPU, as shown in Figure 14. Still, the hash table method performed better in terms of real time, although the results of CPU time were practically the same.

```
                        /* Proc Sql Join Example */
        Proc Sql INOBS=5000000;
          Create Table gdelt_All_V2 as
            Select A.*, B.EventDescription
            From advtbllu.gdelt_all as A, cameo_event_codes as B
            Where A.EventCode=B.EventCode;
 NOTE: Table WORK.GDELT_ALL_V2 created, with 4998980 rows and 10 columns.
        Quit;
 NOTE: PROCEDURE SQL used (Total process time):
        real time              35.58 seconds
        cpu time                8.14 seconds
```

Figure 14. DATA step Merge Table Lookup

To assess the efficiency as the volume of data increases, additional tests were conducted using the GDELT_ALL data set at larger sample sizes of 8 million and 12 million observations. The tests were conducted using each of the three table lookup methods; DATA step merge, PROC SQL join, and hash table lookup. As shown in Table1 below, in real time, the hash table was the most efficient at sample sizes of 5 million and 12 million observations.

	DATA step Merge	Proc Sql Join	hash table
5000000 Obs	52.75	35.58	14.92
8000000 Obs	45.7	16.04	20.41
12000000 Obs	1:54.73	48.25	30.39

Table 1. Real time Performance Comparison: Intermediate methods vs. hash table

The results in CPU time are displayed in Table 2. In CPU time, there was no performance improvement of using the hash table instead of the PROC SQL join. However, there was a large gap between each of these two techniques, and the resource-intensive DATA step merge. This is due in part to running the time-consuming SORT procedure on the base table. The most striking difference between the hash table and DATA step merge came at the sample size of 12 million observations,.

	DATA step Merge	Proc Sql Join	hash table
5000000 Obs	24.75	8.14	7.95
8000000 Obs	37.15	11.47	11.90
12000000 Obs	56.55	18.68	18.41

Table 2. CPU Time Performance Comparison: Intermediate methods vs. hash table

METHOD #3: MULTIPLE SET STATEMENTS

Another method for performing a table lookup is using a DATA step with multiple SET statements. This is conceptually similar to performing a DATA step merge, but the MERGE statement is not used. This allows the programmer to take direct control of the process of matching up records based on the value of one or more key variables. It also has the potential to dramatically reduce processing time when compared with the use of the MERGE statement.

It is important to note that this method still requires the input data sets be sorted. As in the traditional merge, we will be stepping through each data set sequentially, reading each record only once.

A SIMPLE ANALOGY

Imagine you are given two alphabetized stacks of books. Your task is to make a list of the books that are common to both stacks. You take a book from the top of each stack, one in each hand, and compare them. In your left hand is a copy of "Anna Karenina" and in the other is "Animal Farm". Since these do not match, you should move along to the next book from one of the stacks, but which one? We know the stacks are already sorted alphabetically, so it would not make sense to continue taking books from the stack that had "Anna Karenina" on top. That title comes after "Animal Farm" alphabetically. If we continue drawing books from that stack, we will go through the entire stack without finding a match.

Rather, we must discard "Animal Farm" and draw another book from the right-hand stack. If we were to find another copy of "Anna Karenina", we would add that to our list of matches. But what if we find that "Anne of Green Gables" is next on the stack that began with "Animal Farm"? That is also not a match for "Anna Karenina", but we should now draw our next book from the left-hand stack because the last book we took from that stack ("Anna Karenina") comes before the most recent draw from the right-hand stack ("Anne of Green Gables").

This process is analogous to performing a table lookup using two SET statements. The stacks are data sets, the books are records, and the book titles are our key variables. We use two SET statements to read records from the respective input data sets, and we choose which data set to read next by comparing values of our key variable.

EXAMPLE WITH TWO SET STATEMENTS

To illustrate this method, we return to the example used above when the hash object was introduced. We have a large claims data set, CARRIER_CLAIMS_2010, and we wish to lookup the value of a variable called PRFNPI from another data set called FINDER_ATTRIB. The lookup is to be performed using a shared variable, BENE_ID.

The code below shows how this table lookup can be performed using two SET statements. Notice we first sort each data set by the common key variable BENE_ID, just as we would if we were using the MERGE statement.

```
proc sort data=sasfiles.carrier_claim_2010 out=carrier_claim_2010;
  by bene_id;
run;

proc sort data=sasfiles.finder_attrib out=finder_attrib;
  by bene_id;
run;

data bene_match;
  set carrier_claim_2010(rename=(bene_id=bid));
  if bid = bene_id then output;
  do while (bid > bene_id);
    set finder_attrib;
    if bid = bene_id then output;
  end;
run;
```

The SET statement reads one record from the data set(s) specified on the statement each time it is executed. Thus, when this DATA step begins executing, the first SET statement reads in the first record from the CARRIER_CLAIM_2010 data set. As discussed earlier in the paper, these values are held in memory in a storage

location known as the Program Data Vector (PDV). However, because a RENAME= data set option was used in conjunction with the CARRIER_CLAIM_2010 data set on the SET statement, the values of the variable BENE_ID from the data set will be referenced in the PDV using the name BID instead.

The next line is an IF statement which compares the values of BID and BENE_ID. Since we renamed BENE_ID from the claims data set to BID, there will not be a value for BENE_ID in the PDV until a record is read from the FINDER_ATTRIB data set, which has not yet occurred. The IF condition will always be false the first time through the loop. This statement will make more sense in subsequent iterations of the DATA step, so we'll revisit it later.

Next, a DO WHILE loop is encountered. Since BENE_ID is still missing, any non-missing value of BID will make the WHILE condition true. Inside the DO loop, we find the second SET statement. During the first iteration of the DO loop within our first iteration of the DATA step, this SET statement will read the first record from the FINDER_ATTRIB data set and write the values to the PDV.

The DO loop continues iterating until it reads a value of BENE_ID which matches or surpasses the value of BID, at which point the WHILE condition ceases to be true. If the values of BID and BENE_ID are equal, then our table lookup has succeeded and the contents of the PDV are written to the output data set. Regardless of whether BENE_ID matches or exceeds BID, the DO loop terminates and we reach the end of the current iteration of the DATA step.

At that point, assuming additional unread records remain in the claims data set, the flow of execution returns to the top of the DATA step and the next iteration begins. There are now three possibilities regarding the new value of BID: it is either less than, equal to, or greater than the most recently read value of BENE_ID.

1. If the new BID is less than BENE_ID, we know there is no match for BID in FINDER_ATTRIB. Both the IF condition and the WHILE condition will be false and execution will return to the top of the DATA step and we'll move on to the next record from the claims data set.

2. If the new BID is equal to BENE_ID, another match has been found. The IF condition will be true, but the WHILE condition will be false. A record will be written to the output data set, and then execution will return to the top of the DATA step to read the next record from the claims data set.

3. If the new BID is great than BENE_ID, it's time to read more records from FINDER_ATTRIB until the value of BENE_ID "catches up" with BID again. The IF condition will be false, but the WHILE condition will be true. The DO loop will begin reading records from FINDER_ATTRIB again.

This process of reading records continues, alternating between the two data sets, always reading next from the data set that is "further behind" and checking for a potential match with the data set that is "further ahead". When one of the SET statements reaches the end of its corresponding input data set, the DATA step will complete the current iteration and then terminate.

METHOD #4: INDEXES WITH THE KEY= OPTION

Indexes give us yet another method for performing table lookups in SAS. An index provides a way to access a data set based on the value of a key variable. Since this type of access is non-sequential, there is no need to sort the data set first. Before such a table lookup can be performed, an index must first be created on the data set containing the values to be looked up. This is commonly done using the DATASETS procedure as follows.

```
proc datasets;
  modify finder_attrib;
    index create bene_id / unique;
quit;
```

The MODIFY statement is used to specify the data set on which the index is to be created. This is followed by the INDEX CREATE statement, which is subordinate to the MODIFY statement. The INDEX CREATE statement allows us to create either a simple index consisting of one key variable or a composite index consisting of multiple variables. In this example, we have created a simple index on the data set FINDER_ATTRIB using the key variable BENE_ID. The UNIQUE option specifies that values of BENE_ID must be unique or else the index will not be created.

Once the index has been created, it can be used within a SET statement by using the KEY= option to specify the name of the key variable to be used (or, in the case of a composite index, the name of the index itself). The following DATA step code illustrates its use:

```
data bene_match;
  set carrier_claim_2010;
  set finder_attrib key=bene_id/unique;
  if _iorc_ ne 0 then call missing(prfnpi);
run;
```

The first SET statement reads a record from the CARRIER_CLAIM_2010 data set and places those values into the PDV. Note that this data set does not need to be sorted in any particular order, nor does it need to be indexed itself. This has the potential to be tremendously advantageous as sorting and indexing large data sets can be quite costly.

The second SET statement reads a record from the FINDER_ATTRIB data set. However, because the KEY= option is in effect, this is not an ordinary, sequential read. Rather, it uses the index we created earlier in order to directly access the record corresponding with the value of BENE_ID from the PDV. This gives us a very fast way to perform a table lookup.

The final IF statement exists to handle a situation where no record is found in FINDER_ATTRIB corresponding with the value of the key variable BENE_ID that was read from the claims data set. If that happens, the SET statement will set the value of _IORC_ (the input/output return code) to something other than zero. In such a case, we explicity set the value of PRFNPI to missing using the CALL MISSING routine so that the previous value of PRFNPI will be cleared from the PDV before an output record is written.

METHOD #5: ARRAYS AND THE DOW LOOP

One final advanced technique discussed in this paper involves the use of arrays and a special code construct known as the DOW loop. Some have referred to this method as "key-indexing", although it is not related to the approach described above involving indexes and the KEY= option.

THE DOW LOOP

Much has been written about the DOW loop since it was first popularized by (and named for) Ian Whitlock nearly 20 years ago. We briefly describe the concept here, but Paul Dorfman (2009) provides a more thorough overview of the DOW loop and its history.

The DATA step is an implied loop. In a typical DATA step, the statements are executed repeatedly so long as there remain additional records on any input data sets being read. The DOW loop is a coding technique by which the programmer takes control of this looping process. The basic form of the DOW loop consists of a DO loop with a SET statement and an OUTPUT statement contained therein as shown below:

```
data new;
  do until(eof);
    set old end=eof;
    output;
  end;
  stop;
run;
```

During each iteration of the DO loop, the SET statement reads one record from the input data set into the PDV and the OUTPUT statement writes the contents of the PDV to a record in the output data set. This process repeats until the SET statement has read the final record from the input data set. At that point, the EOF variable becomes true and the DO loop terminates. The STOP statement ensures that the DATA step itself does not iterate again.

The result of the above DATA step is no different than if we had simply used the SET statement alone in the ordinary way, without the DO loop or the OUTPUT or STOP statements. However, this construct is very flexible and powerful. Allen (2009) details several practical uses for the DOW loop.

EXAMPLE USING THE DOW LOOP AND ARRAYS

Table lookups can be performed very efficiently using a combination of two DOW loops and a temporary array. The first DOW loop reads through the lookup table and loads all the values into the array indexed on the key variable used to perform the lookup. The second DOW loop reads through the base data set, performs the lookup by accessing the

array based on that same key variable, and writes an output record. In addition to being fast, this method has the significant advantage that neither data set need be sorted. However, because we are using the key variable itself as the array index, this method is only suitable when the key variable is numeric (or can be readily converted to numeric).

Since the Medicare claim examples presented above are based on an alphanumeric key variable, we base this example on the GDELT data set presented earlier in the discussion on hash tables. The GDELT data set has approximately 13.6 million records. One of the variables (EventCode) is a numeric event code. The lookup table, CAMEO_EVENT_CODES, contains these event codes in a variable called CameoEventCode as well as an event description called EventDescription. Using the following DATA step code, we create a new data set based on the GDELT data set with the event descriptions added.

```
data gdelt2;
  array descriptions [2042] $71 _temporary_;
  do until(alldesc);
    set sasfiles.cameo_event_codes end=alldesc;
    descriptions[CameoEventCode] = EventDescription;
  end;
  do until(allevents);
    set sasfiles.gdelt_all end=allevents;
    EventDescription = descriptions[EventCode];
    output;
  end;
  stop;
  drop CameoEventCode;
run;
```

The first thing to notice is the array declaration near the top of the code. This statement declares a temporary array containing 2,042 elements, each of which is a 71-character character string. Temporary arrays consist of unnamed elements that can only be referenced using the array name with an index.

The number of array elements was set at 2,042 because that happens to be the value of the largest event code in the input data set. The event codes themselves will serve as the index to this array. Since the EventDescription variable in the CAMEO_EVENT_CODES data set is character variable with a length of 71, those same attributes are given to the array elements to ensure they are sufficiently large to hold the event description values.

After the array declaration comes the first DOW loop. Inside a DO UNTIL block is a SET statement which reads records from CAMEO_EVENT_CODES, which is the lookup table. After each record is read, an assignment statement copies the value of EventDescription into the array element with the corresponding value of CameoEventCode. This DO loop repeats until all records have been read from the lookup table.

At that point, all the values of EventDescription from the lookup table have been loaded into the array. It's not necessary that all array elements have been used. Unused array elements will not cause any problem, although they do consume memory.

Once the first DOW loop has completed, the second one follows immediately. It also consists of a DO UNTIL block with a SET statement inside. This SET statement reads a record from our base table, GDELT_ALL. In the subsequent assignment statement, the value of EventCode which was read from GDELT_ALL is used as the index in a reference to the temporary array that contains the event description. This array reference returns the event description which had been previously stored in the array during the first DOW loop and stores it in the variable EventDescription. Finally, an OUTPUT statement writes a record to the output data set. This second DOW loop repeats until all records have been read from the base data set and an equal number of records have been written to the output data set.

Everything just described occurred within a single iteration of the DATA step itself. That includes reading all the records from both input data sets and writing all the records to the output data set. Because this was done using the DOW loop, there is no need for the implied looping mechanism built into the DATA step. For this reason, a STOP statement is placed after the second DOW loop to ensure the DATA step loop itself does not begin again.

As mentioned earlier, the main advantages of this method are that it is efficient and does not require sorted data. The primary drawback is that it only works with a numeric key variable. Carpenter (2012, pp. 226-227) presents an

adaptation of this technique for use with a character variable as the key. Another potential drawback is that the amount of memory required to hold the array could become problematic if the number of array elements were to become sufficiently large.

COMPARISON OF ADVANCED TABLE LOOKUP METHODS

Each of the five methods discussed has advantages and disadvantages. The programmer should evaluate the various techniques as they relate to the task at hand to make a suitable selection. There are several factors one might need to consider. Among these are performance, scalability, memory use, simplicity, ease of maintenance, and reusability for future projects. Not all of these will apply to every situation. Here we will focus on comparing performance.

In order to provide a useful comparison, it is necessary to run each method using the same data on the same hardware. Each of the five methods were used to merge the GDELT_ALL data set with the event codes and descriptions in CAMEO_EVENT_CODES. For comparison purposes, an ordinary DATA step merge and a PROC SQL join were also used. The code and log appear in Appendix 2. The results are summarized below.

Method	CPU Time
DATA step merge	29.30 seconds (including both sorts)
PROC SQL join	30.92 seconds
Method #1: PROC FORMAT	21.29 seconds (including creating the format)
Method #2: hash table	18.28 seconds
Method #3: Two SET Statements	28.27 seconds (including both sorts)
Method #4: Index	21.15 seconds (including creating the index)
Method #5: Array and DOW Loop	16.62 seconds

Table 3. Comparison of Table Lookup Methods – Performance Results

Note that all five advanced methods discussed in this paper outperform both the DATA step merge and the PROC SQL join. However, method #3 (two SET statements) is only a slight improvement because it still requires significant time to sort the data sets. The other four methods, which do not require sorted data, are significantly faster. However, the resulting data set remains unsorted. If a sorted output data set is needed anyway, then much of the advantage of these method vanishes.

CONCLUSION

Table lookups are a common operation needed in data processing and analysis. While the standard approaches work well in most situations, it is advantageous for the SAS programmer to be aware of advanced methods that may be beneficial in certain settings. As with many aspects of programming, the optimal method depends on the situation and task at hand. It is often wise to perform some benchmarking to determine which method(s) will best meet the goals for a given job.

REFERENCES

Allen, Richard Read, 2009. "Practical Uses of the DOW Loop." Western Users of SAS Software (WUSS) 2009 Conference. https://www.lexjansen.com/wuss/2009/tut/TUT-Allen.pdf.

Carpenter, Arthur L., 2001. "Table Lookups: From IF-THEN to Key-Indexing." SAS Users Group International 26, Paper 158. https://support.sas.com/resources/papers/proceedings/proceedings/sugi26/p158-26.pdf.

Carpenter, Art. 2012. *Carpenter's Guide to Innovative SAS® Techniques*. Cary, NC: SAS Institute Inc.

Carpenter, Arthur L., 2015. "Table Lookup Techniques: From the Basics to the Innovative." SAS Global Forum 2015, Paper 2219. https://support.sas.com/resources/papers/proceedings15/2219-2015.pdf.

Dorfman, Paul M. and Koen Vyverman, 2009. "The DOW-Loop Unrolled." SAS Global Forum 2009, Paper 038.
http://support.sas.com/resources/papers/proceedings09/038-2009.pdf.

Horstman, Joshua M., 2017. "Beyond IF THEN ELSE: Techniques for Conditional Execution of SAS® Code," SAS Global Forum 2017, Paper 326. https://support.sas.com/resources/papers/proceedings17/0326-2017.pdf.

Horstman, Joshua M., 2018. "Merge with Caution: How to Avoid Common Problems when Combining SAS Datasets." SAS Global Forum 2018, Paper 1746. https://www.sas.com/content/dam/SAS/support/en/sas-global-forum-proceedings/2018/1746-2018.pdf.

ACKNOWLEDGMENTS

CONTACT INFORMATION

Your comments and questions are valued and encouraged. Contact the author at:

Jay Iyengar
Data Systems Consultants LLC
datasyscon@gmail.com
https://www.linkedin.com/in/datasysconsult/

Josh Horstman
Nested Loop Consulting
317-721-1009
josh@nestedloopconsulting.com
www.nestedloopconsulting.com

Jay Iyengar is principal of Data Systems Consultants LLC. He is a SAS Consultant, Trainer, and SAS Certified Advanced Programmer. He is co-leader of the Chicago SAS Users Group, WCSUG. He's presented papers at SAS Global Forum (SGF), Midwest SAS Users Group (MWSUG), Wisconsin Illinois SAS Users Group (WIILSU), Northeast SAS Users Group (NESUG), and Southeast SAS Users Group (SESUG) conferences. He has been using SAS since 1997. His industry experience includes Healthcare, Pharmaceutical, Public Health, Database Marketing Educational Testing, and International Trade.

Josh Horstman is an independent statistical programming consultant and trainer based in Indianapolis with over 20 years' experience using SAS in the life sciences industry. He specializes in analyzing clinical trial data, and his clients have included major pharmaceutical corporations, biotech companies, and research organizations. Josh is a SAS Certified Advanced Programmer who loves coding and is a frequent presenter at SAS Global Forum and other SAS User Group meetings.

TRADEMARK CITATION

Data Set Name	ADVTBLLU.GDELT_ALL	Observations	13631475
Member Type	DATA	Variables	9
Engine	V9	Indexes	0
Created	08/13/2019 17:24:02	Observation Length	72
Last Modified	08/13/2019 17:24:02	Deleted Observations	0
Protection		Compressed	NO
Data Set Type		Sorted	NO
Label			
Data Representation	SOLARIS_X86_64, LINUX_X86_64, ALPHA_TRU64, LINUX_IA64		
Encoding	utf-8 Unicode (UTF-8)		

Engine/Host Dependent Information	
Data Set Page Size	65536
Number of Data Set Pages	15013
First Data Page	1
Max Obs per Page	908
Obs in First Data Page	864
Number of Data Set Repairs	0
Filename	/folders/myfolders/AdvTblLkup/gdelt_all.sas7bdat
Release Created	9.0401M6
Host Created	Linux
Inode Number	131490
Access Permission	rw-rw-r--
Owner Name	sasdemo
File Size	938MB
File Size (bytes)	983957504

	Alphabetic List of Variables and Attributes					
#	Variable	Type	Len	Format	Informat	Label
3	Actor1Code	Char	15	$15.	$15.	Actor1Code
4	Actor1EthnicCode	Char	3	$3.	$3.	Actor1EthnicCode
5	Actor1Type1Code	Char	3	$3.	$3.	Actor1Type1Code
6	Actor2Code	Char	15	$15.	$15.	Actor2Code
7	Actor2EthnicCode	Char	3	$3.	$3.	Actor2EthnicCode
8	Actor2Type1Code	Char	3	$3.	$3.	Actor2Type1Code
2	Day	Num	8	BEST.		Day
9	EventCode	Num	8	BEST.		EventCode
1	GlobalEventID	Num	8	BEST.		GlobalEventID

APPENDIX II – LOG OUTPUT FROM COMPARISON OF METHODS

```
171  /*** FOR COMPARATIVE PURPOSES - data step merge ***/
172
173  proc sort data=sasfiles.gdelt_all out=gdelt_all_sorted;
174      by eventcode;
175  run;

NOTE: There were 13631475 observations read from the data set SASFILES.GDELT_ALL.
NOTE: The data set WORK.GDELT_ALL_SORTED has 13631475 observations and 9 variables.
NOTE: PROCEDURE SORT used (Total process time):
      real time           42.39 seconds
      cpu time            23.25 seconds

176  proc sort data=sasfiles.cameo_event_codes out=cameo_event_codes_sorted;
177      by cameoeventcode;
178  run;

NOTE: There were 290 observations read from the data set SASFILES.CAMEO_EVENT_CODES.
NOTE: The data set WORK.CAMEO_EVENT_CODES_SORTED has 290 observations and 2 variables.
NOTE: PROCEDURE SORT used (Total process time):
      real time           0.07 seconds
      cpu time            0.01 seconds

179  data gdelt_merge;
180      merge gdelt_all_sorted(in=a)
181      cameo_event_codes_sorted(rename=(cameoeventcode=eventcode));
182      by eventcode;
183      if a;
184  run;

NOTE: There were 13631475 observations read from the data set WORK.GDELT_ALL_SORTED.
NOTE: There were 290 observations read from the data set WORK.CAMEO_EVENT_CODES_SORTED.
NOTE: The data set WORK.GDELT_MERGE has 13631475 observations and 10 variables.
NOTE: DATA statement used (Total process time):
      real time           23.32 seconds
      cpu time            6.04 seconds

185
186  /*** FOR COMPARATIVE PURPOSES - proc sql join ***/
187  proc sql;
188    create table gdelt_join as
189      select * from sasfiles.gdelt_all a
190      left join sasfiles.cameo_event_codes b
191      on a.eventcode = b.cameoeventcode;
NOTE: Table WORK.GDELT_JOIN created, with 13631475 rows and 11 columns.

192  quit;
NOTE: PROCEDURE SQL used (Total process time):
      real time           1:08.29
      cpu time            30.92 seconds

196  /*** Method 1 - PROC FORMAT ***/
197
198  data cameo_event_codes;
199      set sasfiles.cameo_event_codes
200          (rename=(cameoeventcode=start eventdescription=label));
201      retain fmtname 'evntcd' type 'N';
```

```
202      keep type start label fmtname;
203  run;


NOTE: There were 290 observations read from the data set SASFILES.CAMEO_EVENT_CODES.
NOTE: The data set WORK.CAMEO_EVENT_CODES has 290 observations and 4 variables.
NOTE: DATA statement used (Total process time):
      real time           0.03 seconds
      cpu time            0.01 seconds


204
205  proc format cntlin=cameo_event_codes;
NOTE: Format EVNTCD has been output.
205!                                        run;


NOTE: PROCEDURE FORMAT used (Total process time):
      real time           0.01 seconds
      cpu time            0.00 seconds


NOTE: There were 290 observations read from the data set WORK.CAMEO_EVENT_CODES.


206
207  data gdelt01;
208      set sasfiles.gdelt_all;
209      eventdescription = put(eventcode,evntcd.);
210  run;


NOTE: There were 13631475 observations read from the data set SASFILES.GDELT_ALL.
NOTE: The data set WORK.GDELT01 has 13631475 observations and 10 variables.
NOTE: DATA statement used (Total process time):
      real time           32.02 seconds
      cpu time            21.28 seconds


211
212  /*** Method 2 - hash object ***/
213
214  data gdelt02(rename=(cameoeventcode=eventcode) drop=rc);
215      if 0 then set sasfiles.cameo_event_codes;
216      if _n_=1 then do;
217          declare hash events (dataset: 'sasfiles.cameo_event_codes');
218          events.definekey ('cameoeventcode');
219          events.definedata('eventdescription');
220          events.definedone ();
221      end;
222      do until (eof1);
223          set sasfiles.gdelt_all
224              (rename=(eventcode=cameoeventcode)) end=eof1;
225          rc = events.find();
226          if rc=0 then output;
227      end;
228  run;


NOTE: There were 290 observations read from the data set SASFILES.CAMEO_EVENT_CODES.
NOTE: There were 13631475 observations read from the data set SASFILES.GDELT_ALL.
NOTE: The data set WORK.GDELT02 has 13629110 observations and 10 variables.
NOTE: DATA statement used (Total process time):
      real time           27.33 seconds
      cpu time            18.28 seconds
```

```
230  /*** Method 3 - double set statement ***/
231
232  proc sort data=sasfiles.gdelt_all out=gdelt_all_sorted;
233      by eventcode;
234  run;
```

NOTE: There were 13631475 observations read from the data set SASFILES.GDELT_ALL.
NOTE: The data set WORK.GDELT_ALL_SORTED has 13631475 observations and 9 variables.
NOTE: PROCEDURE SORT used (Total process time):
      real time   33.30 seconds
      cpu time   23.81 seconds

```
235  proc sort data=sasfiles.cameo_event_codes out=cameo_event_codes_sorted;
236      by cameoeventcode;
237  run;
```

NOTE: There were 290 observations read from the data set SASFILES.CAMEO_EVENT_CODES.
NOTE: The data set WORK.CAMEO_EVENT_CODES_SORTED has 290 observations and 2 variables.
NOTE: PROCEDURE SORT used (Total process time):
      real time   0.04 seconds
      cpu time   0.01 seconds

```
238  data gdelt03;
239      set gdelt_all_sorted;
240      if eventcode = cameoeventcode then output;
241      do while (eventcode > cameoeventcode);
242          set cameo_event_codes_sorted;
243          if eventcode = cameoeventcode then output;
244      end;
245  run;
```

NOTE: There were 13631475 observations read from the data set WORK.GDELT_ALL_SORTED.
NOTE: There were 288 observations read from the data set WORK.CAMEO_EVENT_CODES_SORTED.
NOTE: The data set WORK.GDELT03 has 13629110 observations and 11 variables.
NOTE: DATA statement used (Total process time):
      real time   17.49 seconds
      cpu time   5.51 seconds

```
246
247  /*** Method 4 - index ***/
248  proc copy in=sasfiles out=work;
249      select gdelt_all cameo_event_codes;
250  run;
```

NOTE: Copying SASFILES.GDELT_ALL to WORK.GDELT_ALL (memtype=DATA).
NOTE: There were 13631475 observations read from the data set SASFILES.GDELT_ALL.
NOTE: The data set WORK.GDELT_ALL has 13631475 observations and 9 variables.
NOTE: Copying SASFILES.CAMEO_EVENT_CODES to WORK.CAMEO_EVENT_CODES (memtype=DATA).
NOTE: There were 290 observations read from the data set SASFILES.CAMEO_EVENT_CODES.
NOTE: The data set WORK.CAMEO_EVENT_CODES has 290 observations and 2 variables.
NOTE: PROCEDURE COPY used (Total process time):
      real time   22.64 seconds
      cpu time   15.35 seconds

```
251  proc datasets;
252      modify cameo_event_codes;
253          index create cameoeventcode / unique;
```

```
NOTE: Simple index CAMEOEVENTCODE has been defined.
254  quit;

NOTE: MODIFY was successful for WORK.CAMEO_EVENT_CODES.DATA.
NOTE: PROCEDURE DATASETS used (Total process time):
      real time            0.24 seconds
      cpu time             0.04 seconds


256  data gdelt04(rename=(cameoeventcode=eventcode));
257      set gdelt_all(rename=(eventcode=cameoeventcode));
258      set cameo_event_codes key=cameoeventcode/unique;
259      if _iorc_ ne 0 then call missing(eventdescription);
260  run;
NOTE: There were 13631475 observations read from the data set WORK.GDELT_ALL.
NOTE: The data set WORK.GDELT04 has 13631475 observations and 10 variables.
NOTE: DATA statement used (Total process time):
      real time            32.39 seconds
      cpu time             21.11 seconds


262  /*** Method 5 - arrays and DOW loop ***/
263
264  data gdelt05;
265      array descriptions [2042] $71 _temporary_;
266      do until(alldesc);
267          set sasfiles.cameo_event_codes end=alldesc;
268          descriptions[CameoEventCode] = EventDescription;
269      end;
270      do until(allevents);
271          set sasfiles.gdelt_all end=allevents;
272          EventDescription = descriptions[EventCode];
273          output;
274      end;
275      stop;
276      drop CameoEventCode;
277  run;

NOTE: There were 290 observations read from the data set SASFILES.CAMEO_EVENT_CODES.
NOTE: There were 13631475 observations read from the data set SASFILES.GDELT_ALL.
NOTE: The data set WORK.GDELT05 has 13631475 observations and 10 variables.
NOTE: DATA statement used (Total process time):
      real time            26.52 seconds
      cpu time             16.62 seconds
```