

Look Up Not Down: Advanced Table Lookups in Base SAS®

Jay Iyengar, Data Systems Consultants

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 I concentrate on advanced table lookup methods such as formats, and HASH objects. I conceptually examine what advantages they provide the SAS programmer over basic methods. I 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 My 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, I 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 data set. 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 codes to indicate medical diagnoses or procedures. They must be linked to smaller reference tables to obtain the code definitions. Figure 1 displays a schematic of a table lookup with five tables.

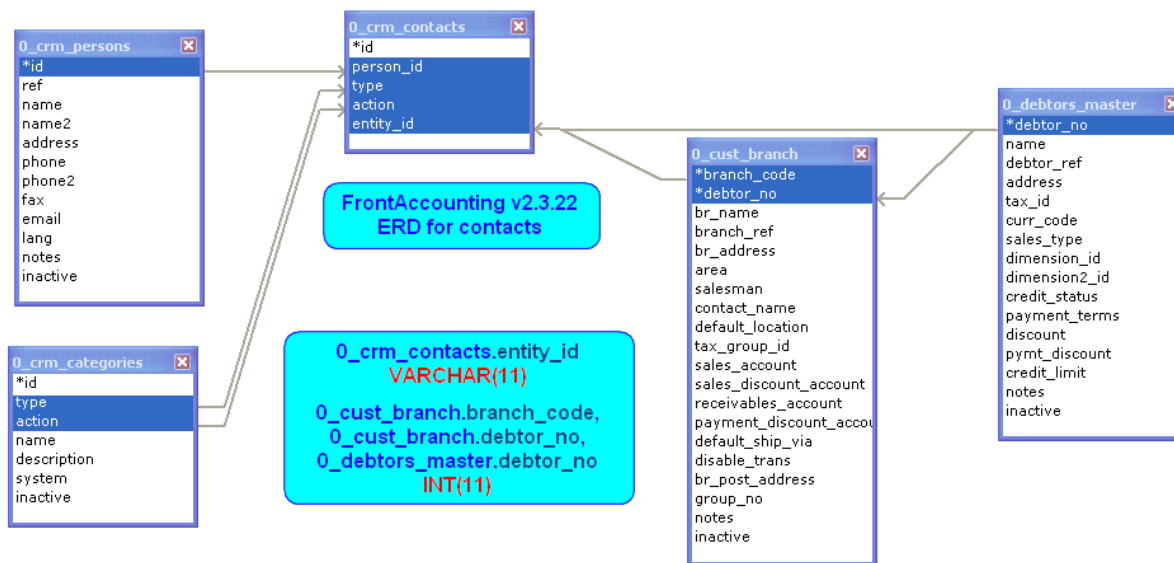


Figure 1. Table Lookup Schema

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 data sets, 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, I 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 CHOOSE/CHOOSE 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:

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 and basic ways to perform a table lookup is to use SAS Formats. Although this is a basic technique, I've chosen to explore FORMATS because they present an alternative to the commonly used DATA STEP merge and PROC SQL join. Thus, for the purposes of this paper, FORMATS will be treated as an advanced lookup technique.

To perform a table lookup using formats, the SAS user creates a user-defined format using PROC FORMAT. The programmer has two options. 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 variables in a SAS data set. Thus, the lookup table may not be a table at all depending on the method chosen. If the programmer creates the format from a SAS data set, the data set must have a specific structure containing key values and data values. Conversely, the lookup values need to be specified 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. RACEETH assigns descriptive labels for racial categories to numeric values. The format is defined as numeric because the key values are numeric, which are assigned to character data values. 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 I stated earlier, administrative healthcare data sets contain codes to indicate diagnoses, procedures, physicians, etc.

In this case, the format \$RACEETH contains the lookup values, instead of the values being stored in a separate 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 included as an argument in the PUT function and applied to RACE.

One drawback of this technique is that it requires a lot of maintenance for larger sets of data values. A demographic variable such as race is well-suited to this technique, because it contains a small number of distinct values. 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 that the values need to be updated, or values need to be added to the format. In this case, the programmer needs to manually modify and edit the hard-coded values. This takes the risk that typos or mistakes will be made which might compromise the validity of the data.

In this example, the format with a PUT function method was executed efficiently. In terms of performance, the table lookup processed in a very short amount of time with a base table containing over a 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 technique.

PROC FORMAT CNTLIN SAS DATA SET

In the previous example, I 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 contain several variables. START contains the values in a VALUE statement from the left side of the equal's 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 advtblu.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 advtblu.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. PROC FORMAT CNTLIN – Applying format using PUT function

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. This 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

WHAT ARE 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, while data values are accessed and retrieved from the table. 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 'Hash Table Name' ("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 which the hash table is based on.

```
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 I'm interested in. The data variable is specified in parentheses and double-quotes and can be taken from the SAS data set which populates the hash table. 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;
```

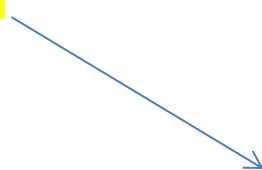


PDV						
BENE_ID	PRFNPI	EOF1	CLM_ID	RC	_ERROR_	_N_
		0		.	0	1

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 variable _N_ 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 routine executes

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 the 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 which contains system resources. The virtual machine has 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 advtblu.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 ADVTBLU.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 5000000. The lookup table with event code 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 Figure 12, the SAS log shows 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=advtblu.gdelt_all(Ob=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 = advtblu.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 advtblu.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

CONCLUSION

Table lookups are a common operation needed in data processing and manipulation. 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>.
- SAS Institute Inc., 2010. "SAS Certification Prep Guide – Advanced Programming for SAS 9." Cary, NC: SAS Institute Inc.
- ACL. "Using ACL Analytics>Preparing data for Analysis>Combining Data". Accessed August 9, 2018. https://enablement.acl.com/helpdocs/analytics/13/user-guide/en-us/Content/da_cmbining_data/combining_data.htm?TocPath=Combining%20data|_____

ACKNOWLEDGMENTS

The author would like to thank Jim Blum, SESUG 2024 Academic Chair, Lucia Alexander, SESUG 2024 Operations Chair, Mark Jordan and Jak Andan, Intermediate and Advanced SAS Skills Section Co-Chairs, and the SESUG 2024 Executive Committee and Conference Team for accepting my abstract and paper and for organizing this great conference.

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/>

Jay Iyengar is director of Data Systems Consultants LLC. He is a SAS consultant, trainer, and SAS Certified Advanced Programmer. He was co-leader and organizer of the Chicago SAS Users Group (WCSUG) from 2015-19. He's presented papers and training seminars at SAS Global Forum (SGF), and many other regional, local and industry-specific SAS users group conferences and meetings. His main industry experience is in Health care, Public Health and Pharmaceutical. He obtained his Bachelor's in Public Policy and Economics from Syracuse University and his master's degree from the American University.

TRADEMARK CITATION

SAS and all other SAS Institute Inc. product or service names are registered trademarks or trademarks of SAS Institute Inc. in the USA and other countries. ® indicates USA registration.

Other brand and product names are trademarks of their respective companies.

APPENDIX I – GDELT DATA SET

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 – SAS LOG 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.GDELTA

NOTE: The data set WORK.GDELTA_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.GDELTA_SORTED.

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

NOTE: The data set WORK.GDELTA_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
```