Paper 1429-2017

Yet Another Lookup Method: Updatable Indexed Data Sets

Jack Hamilton, Kaiser Permanente

.

ABSTRACT

SAS ® has many methods of doing table lookups in data steps: formats, arrays, hash objects, the SASMSG function, indexed data sets, and so forth. Of those, hash objects and indexed data sets allow you to specify multiple lookup keys and return multiple table values. Both can be updated dynamically in the middle of a data step as you obtain new information (such as reading new keys from an input file, or creating new synthetic keys). Hash objects are very flexible and fast, and fairly easy to use, but are limited to the amount of data that can be held in memory. Indexed data sets may be slower, but are not limited to what can fit into memory, and so in some circumstances may be your only option. This presentation discusses how to use an indexed data set for table lookup and update it dynamically using the MODIFY statement and its allies.

INTRODUCTION

Almost all SAS users, whether beginning, intermediate, or advanced, use lookups in their programs. A lookup is basically a kind of mapping: it takes one or more inputs and returns one or more outputs, according to a fixed set of values. It does not calculate a new value, as a function might; it merely "looks up" a value in an existing table.

An example of a lookup is taking the state code "PR" and converting it to "Puerto Rico".

This paper shows some commonly used lookup techniques (formats, merging, joins), some less commonly used ones (macro values, the SASMSG function), and the topic of this manuscript, indexed data sets.

The code used in this paper may be downloaded from sascommunity.org.

SAMPLE DATA

This paper uses US ZIP codes for the lookup table. Data come from the SASHELP.ZIPCODE data set provided with base SAS. To make the data easier to work with, not all rows and columns are included, and ZIP codes are converted from numeric to character.

The data creation, along with some other simplifying steps, are illustrated in the following code::

```
/* Get ZIP codes for California and New York */
data zip_master;
    set sashelp.zipcode (keep=zip city statecode city county countynm
                         rename=(zip=zip_n statecode=state));
    where state in ('CA', 'NY');
    zip = put(zip_n, z5.);
    drop zip_n;

run;

/** Demonstrate that there's only one record for each zip code.  **/
proc sort data=zip_master nodupkey force;
    by zip;
run;


/* To make this example easier, keep only one ZIP code for each city.  */
```

```
proc sort data=zip_master nodupkey force;
    by state city zip;
run;

data zip_master;
    set zip_master;
    by state city zip;
    if first.city;
run;
```

A later example will need data from an additional state, which are first stored in a separate data set. To illustrate a different method of extraction, this is done with the SQL procedure:

```
/** For later use, get ZIPs for KY. */
proc sql;
    create table zip_master_ky as
        select
            put(zip, z5.0) as zip,
            city, statecode as state, county, countynm
        from
            sashelp.zipcode
        where
            statecode = 'KY'
        group by
            state, city
        having
            zip = min (zip);

quit;
```

We also need data sets contained values that will be looked up. The first data set contains only ZIP codes; later lookup examples will lookup this values in the lookup table to obtain additional values:

```
/* Some items to look up, some that work, some that don't. */
data lookup_by_zip;
    length zip $5;
    do zip = '12456',   /* Mount Marion     */
             '95975',   /* Rough & Ready    */
             '34567';   /* Not in lookup    */
        output;
    end;
    stop;
run;
```

We will also need a data set with more complicated values for multi-value lookups. In this case, we will use ZIP code, state, and city:

```
 data lookup_by_state_city;
    if 0 then
        set zip_master (keep=state city);
    do state = 'CA';
        do city = 'The Sea Ranch',
                  'Freedom',
                  'Coarsegold',
                  'Erewhon';        /* Doesn't exist    */
            output;
```

```
        end;
    end;
    do state = 'NY';
        do city = 'Ardsley on Hudson',
                  'Feura Bush',
                  'Freedom',
                  'Erewhon';
            output;
        end;
    end;
    state = 'KY';
    city = 'Eighty Eight';    /* Real, but not in lookup table */
    output;
    stop;
run;
```

LOOKUP USING FORMATS

Formats are one of the most powerful and ubiquitous features in base SAS. The FORMAT procedure is used to create a format from explicit code or from an input data set, and a later part of the program uses the format to convert an input value to an looked-up output value. Formats can be used in data steps, procedures, macros, and SQL, and in some cases can even be embedded into database systems for faster processing. SAS software ships with many useful formats. Most formats are just lookups; it is also possible to create formats that perform algorithms to calculate values, but that is not common, and also is not an example of lookup and so it is not shown here.

This example creates a format converting a ZIP code to a city name:::

```
/** Create a format from the master with zip as value**/
data zip_cntlin (keep=fmtname start label hlo);
    retain fmtname '$lookup_by_zip';
    set zip_master end=end;
    start = zip;
    label = city;
    output;
    if end then
        do;
        hlo = 'o';
        label = '??';
        output;
        end;
run;
proc format cntlin=zip_cntlin;
run;
```

By default, if a format does not contain a mapping for an input value, it returns the original input value. It is often better to return a different value for a failed lookup, and the code starting with "if end then" shows how to do that.

Doing the lookup in a data step is as simple as using the PUT function:

```
data fmt_lkup_city_from_zip;
    set lookup_by_zip;
    city = put(zip, $lookup_by_zip.);
run;
```

It is also possible to look up a single output value from two input values by simply concatenating the values together. This is easiest if one of the values is of fixed length:

```
/** Create a format from the master with zip as value**/
data zip_cntlin (keep=fmtname start label hlo);
    retain fmtname '$lookup_by_state_city';
    set zip_master end=end;
    start = cat(state, city);
    label = zip;
    output;
    if end then
        do;
        hlo = 'o';
        label = '??';
        output;
        end;
run;

/* Create the format */
proc format cntlin=zip_cntlin;
run;

data fmt_lkup_zip_from_state_city;
    set lookup_by_state_city;
    zip = put(cat(state, city), $lookup_by_state_city.);
run;
```

It's possible to look up multiple output values from a single input using a similar technique. An alternative is to use a separator character not used in the data; that is helpful when the output values are of irregular lengths. Both techniques are shown in the following code:

```
/** Create a format from the master with state city as value**/
data zip_cntlin (keep=fmtname start label hlo);
    retain fmtname '$lookup2_by_zip';
    set zip_master end=end;
    start = zip;
    label = state || city;
    /* One alternative is to use separator character not    */
    /* found in data, e.g.                                  */
    /*     label = catx('|', state, city);                  */
    /* Use of unprintable characters such as '00'x may      */
    /* reduce the number of possible entries.               */
    output;
    if end then
        do;
        hlo = 'o';
        label = '??';
        output;
        end;
run;

proc format cntlin=zip_cntlin;
run;

data fmt_lkup_state_city_from_zip;
    if 0 then set zip_master (keep=state city);
```

```
    set lookup_by_zip;
    __value = put(zip, $lookup2_by_zip.);
    drop __value;
    state = substr(__value, 1, 2);
    city = substr(__value, 3);
    /* If you had used separator character:                */
    /*    state = scan(__value, 1, '|');                   */
    /*    city = scan(__value, 2, '|');                    */
run;
```

It is possible to have multiple values on both input and output, but that gets cumbersome and is not shown here.

DATA STEP MERGE

The MERGE statement in the data step is not always thought of as a lookup technique, but often it is. The big advantage of the MERGE technique is that it can handle any number of lookup keys and output values. The big disadvantage is that both data sets must be sorted by the lookup keys. If the tables are large, this can be expensive.

Here's an example looking up city and state from ZIP:

```
proc sort data=lookup_by_zip;
    by zip;
run;

proc sort data=zip_master out=zip_master_by_zip;
    by zip;
run;

data ds_lkup_state_city_from_zip;
    merge lookup_by_zip (in=want)
          zip_master_by_zip (keep=zip city);
    by zip;
    if want;
    if city = ' ' then
        city = '??';
run;

/* Re-sort required to get the original zip order.  */
proc sort data=ds_lkup_state_city_from_zip;
    by zip;
run;
```

[NOTE: In some versions of the sample code, one of the data set names is incorrect. The code shown above is correct. This also applies to some examples below.]

The other direction, looking up ZIP from city and state, is hardly different:

```
/* Get ZIP from city and state */
proc sort data=lookup_by_state_city;
    by state city;
run;

data ds_lkup_zip_from_state_city;
    merge lookup_by_state_city (in=want)
          zip_master (keep=zip city state);
    by state city;
```

```
    if want;
    if zip = ' ' then
        zip = '??';
run;;
```

You can return additional values if you wish; the sample code contains an example of that.

SQL

The LEFT JOIN in SQL is ideal for this kind of lookup. Use the COALESCE function to fill in the special value for failed lookups. This example uses the SQL procedure, but the same technique would work in pass-through SQL to many database systems.

```
proc sql;
    create table sql_lkup_state_city_from_zip as
        select
            want.*  ,
            coalesce(have.state, '??') as state ,
            coalesce(have.city, '??') as city
        from
            lookup_by_zip as want
        left join
            zip_master as have
        on
            want.zip = have.zip
        order by
            zip;

    create table sql_lkup_zip_from_state_city as
        select
            have.state  ,
            have.city   ,
            coalesce(have.zip, '??') as zip
        from
            lookup_by_state_city as want
        left join
            zip_master as have
        on
            want.state = have.state
                and
            want.city = have.city
        order by
            have.zip;
quit;
```

Note that you don't know how SQL will implement this. It might sort both tables, or it might create a hash table containing the smaller data set, or it might do something else.

ARRAYS

Data step arrays were once a popular way to do certain types of lookups. They have mostly fallen out of use in newly written programs because hash objects are both more flexible and more powerful. Nevertheless, you might see them in older programs, or even find a new use for them.

The usual framework is to read through the lookup data to see how many array entries are needed, and then, in a separate step, read the data into arrays. The main body of the data step then iterates through the arrays each time a lookup needs to be done. The example below uses SQL to find the number of array elements needed and the width of character elements:

```
proc sql noprint;
    select
        count(*)    ,
        max(lengthm(zip)) ,
        max(lengthm(city))
    into
        :__zip_master_obs_ct  ,
        :__zip_lengthm  ,
        :__city_lengthm
    from
        zip_master;
quit;
```

In the main data step, the arrays are initialized in the first iteration. In this case, there are two arrays, one containing the key to be looked up (array __zip) and one containing the values to be be returned (__city). Then the program loops through the key array every time it needs to do a lookup:

```
data array_lkup_city_from_zip;

    array __zip(&__ZIP_MASTER_OBS_CT.)  $&__ZIP_LENGTHM.  _temporary_;
    array __city(&__ZIP_MASTER_OBS_CT.) $&__CITY_LENGTHM. _temporary_;

    if _n_ = 1 then
        do _i_ = 1 to &__ZIP_MASTER_OBS_CT.;
        set zip_master (keep=zip city);
        __zip{_i_} = zip;
        __city{_i_} = city;
        end;

    set lookup_by_zip;

    __found = 0; drop __found;
    do _i_ = 1 to &__ZIP_MASTER_OBS_CT. while (__found=0);
        if __zip{_i_} = zip then
            __found = 1;
    end;

    if __found then
        city = __city{_i_-1};
    else
        city = '??';

    drop _i_;

run;
```

This technique could be expanded to multiple lookup keys and returned values.

An optimization is possible with certain types of data:

If the lookup key is numeric with a small range of values, you can use the lookup key as the array index. For example, if your key values are dates in 2016 and the return value is the name of the doctor on call, you might code something like this:

```
data _null_;

    array doc_on_duty( %sysfunc(inputn(01jan2016, date9.)) :
                            %sysfunc(inputn(31dec2016, date9.)))
                      $50 _temporary_;

    doc_on_duty('30mar2016'd) = 'Garcia';

    doc_date = '16aug2016'd;
    doc_name = doc_on_duty(doc_date);
    if doc_name = ' ' then doc_name = '??';
    put doc_date= doc_name=;

    doc_date = '30mar2016'd;
    doc_name = doc_on_duty(doc_date);
    if doc_name = ' ' then doc_name = '??';
    put doc_date= doc_name=;

run;
```

Of course, you have to be careful to make sure you don't attempt to look up values outside the range covered in the array, or you will get an error.

Note that the data are kept in _temporary_ arrays, which have several advantages over regular arrays. _temporary_ arrays are kept in contiguous memory, which reduces paging while looping. They are automatically RETAINed, and are not in the Program Data Vector, reducing overhead.

HASH OBJECTS (ASSOCIATIVE ARRAYS)

Hash objects are relatively new to SAS, and have replaced many other lookup techniques because of their speed and power. You can think of a hash object as a special kind of array, where the array keys can be strings or a numbers, and don't have to be contiguous. That's why hash objects are also known as associative arrays ("hash object" is actually a description of the implementation. SAS could change the implementation from a hash to a binary tree, and it would be pretty much transparent to the user). Also, unlike regular arrays, hash objects can change in size over the course of the execution of the data step.

Hash objects are adequately documented elsewhere, so I will give only two simple examples without explanation - sometimes having a complete piece of working code in front of your is more valuable than documentation.

In the first example, a hash object is created, and values are loaded from a SAS data set. Creating and populating the hash object is complicated, but using it is very simple - just a simple method call. This code looks up the ZIP code given a city and state:

```
data hash_lkup_zip_from_state_city;

    /* Variable lengths should be declared before adding to     */
    /* hash object, otherwise numeric is assumed.               */
    if 0 then
        set zip_master (keep=zip state city);

    declare associativearray zip_master
                (dataset: "zip_master (keep=city state zip)");
```

```
    __rc = zip_master.DefineKey ('state', 'city');
    __rc = zip_master.DefineData('zip');
    __rc = zip_master.DefineDone ();

    do until (nomore);
        set lookup_by_state_city end=nomore;
        __rc = zip_master.find();
        if __rc ne 0 then
            zip = '??';
        output;
    end;

    stop;

    drop __rc;

run;
```

The second example shows that you can entries to the hash object after it has been created:

```
/* After initial load, add some new entries.    */
data hash_lkup_zip_from_state_city_ky;

    /* Variable lengths should be declared before adding to     */
    /* hash object, otherwise numeric is assumed.               */
    if 0 then
        set zip_master (keep=zip state city);

    /* Load base data set */
    declare associativearray zip_master
                (dataset: "zip_master (keep=city state zip)");
    __rc = zip_master.DefineKey ('state', 'city');
    __rc = zip_master.DefineData('zip');
    __rc = zip_master.DefineDone ();

    /* Load additional values from a data set               */
    /* Make sure to bring in only the variables you need;   */
    /* any other variables will have values retained from   */
    /* the last observation.                                */
    do until (nomore_adds);
        set zip_master_ky (keep=zip city state) end=nomore_adds;
        zip_master.add();
    end;

    /* Do another lookup, which will find a value   */
    /* for the city in Kentucky.                    */
    do until (nomore);
        set lookup_by_state_city end=nomore;
        __rc = zip_master.find();
        if __rc ne 0 then
            zip = '??';
        output;
    end;

    stop;

    drop __rc;
```

```
run;
```

You might notice that both examples use a DOW loop to bring in the records to be looked up. This simplifies the coding, but requires a STOP statement at the end.

MACRO VARIABLES

You can use macro variables as a kind of array, with part of the macro name being the array name and part of it being the key value. You can use either numbers or a restricted set of characters as the value part of the name. The primary use of this technique would be in macros; using it in a data step would not be efficient.

For example, the macro equivalent of "myarray(12) = 13;" would be "%let myarray_12 = 13;". The code below shows several examples of using macro variables:

```
/* You might want to do a lookup in the macro environment.  That's  */
/* not common, but could be useful in table-driven programs.        */


/* First approach: put key value in the variable name.  This works  */
/* only if key values can be part of a valid SAS name.  Use a       */
/* unique prefix to make variables easy to find and to reduce       */
/* collisions with other macro variables that might be hanging      */
/* around.                                                          */

/* I'll use the prefix _ZIP_ to do a ZIP to City lookup.  This      */
/* isn't a good real-life case, but I'll stick to the same data.    */

/* It is possible to read the zip_master data set and create the    */
/* macro variables entirely in the macro language, but a data step  */
/* is much easier - don't have to use data set info functions,      */
/* and don't have to worry as much about macro quoting.             */

data _null_;

    do until (nomore);
        set zip_master end=nomore;
        call symputx('_ZIP_' || zip, city, 'G');
    end;

run;

%put NOTE: &=_ZIP_96103;


/* A more complicated case is when the key values contain spaces or */
/* special characters.  In that case, you can create two parallel   */
/* sets of variables, similar to the array example.                 */
/* I will use State/City to ZIP for this example.                   */

data _null_;

    do i = 1 by 1 until (nomore);
        set zip_master end=nomore;
        call symputx('_STATE_CITY_' || left(put(i, 5.0)),
```

```
                    state || city, 'G');
        call symputx('_ZIPN_' || left(put(i, 5.0)),
                    zip, 'G');
    end;

    call symputx('_STATE_CITY_MAX', i, 'G');

run;

%put NOTE: &=_STATE_CITY_1234 , &=_ZIPN_1234;

/* The lookup is more complicated and has to be done inside a   */
/* macro because it uses a DO loop.                             */
/* Some special characters will require additional macro        */
/* quoting to prevent errors.                                   */

%macro _state_city_zip_lookup(state, city);

    %local i zip found;

    %let found = 0;
    %do i = 1 %to &_STATE_CITY_MAX.;
        %if "&STATE.&CITY." = "&&_STATE_CITY_&I." %then
            %do;
            %let found = 1;
            %goto exit_loop;
            %end;
    %end;

    %EXIT_LOOP:
    %if &FOUND. %then
        %do;
        &&&_ZIPN_&I.
        %end;
    %else
        %do;
        ??
        %end;

%mend;

%put NOTE: Coarsegold, CA is %_state_city_zip_lookup(CA, Coarsegold);

%put NOTE: 88, KY is %_state_city_zip_lookup(KY, 88);
```

THE SASMSG FUNCTION

SASMSG is a little known but documented function that looks up values in a data set. It's a bit difficult to think of a problem to which this would be the best solution, but it's always good to have another tool in your toolbox.

This code shows the use of the function in the data step and the SQL procedure:

```
/* The SASMSG function is intended for the internationalization */
/* of text strings, but if you create an appropriate input data */
/* set, you can do other things with it.  Unfortunately, the    */
/* key field must be in upper case.                             */
```

```
options locale="en_US"; /* required */


proc sql;

    create table state_city_to_zip as
        select
            "en_US"             as locale length=5,
            upcase(state||city) as key length=60,
            1                   as lineno length=5,
            zip                 as text length=1200
        from
            zip_master
        order by
            locale,
            key,
            lineno descending;

    create index indx on state_city_to_zip(locale, key);

quit;

data msg_lkup_zip_from_state_city;

    length __msg $200; drop __msg;

    set lookup_by_state_city;

    __msg = sasmsg('state_city_to_zip', upcase(state || city), 'noquote');

    if __msg ne upcase(state || city) then
        zip = __msg;
    else
        zip ='??';

run;

proc sql;

    create table sql_msg_lkup_zip_from_state_city (drop=__zip) as
        select
            main.* ,
            sasmsg('state_city_to_zip', upcase(state || city), 'noquote')
as __zip ,
            case calculated __zip
                when upcase(state || city) then '??'
                else calculated __zip
                end as ZIP length=5
        from
            lookup_by_state_city as main;

quit;
```

The key is restricted to a 60 uppercase character value.

INDEXED DATA SETS

Indexed data sets are another way to look up data. Indexing provides random access to records in a data set based on a key.

The first thing you need to use indexed data sets, is, of course, indexes. They can be created by the DATASETS procedure, proc SQL, or data set options. I prefer proc SQL because I think it has the simplest and most obvious syntax:

```
proc sql noerrorstop;
    create unique index zip
        on zip_master(zip);
    create unique index state_city
        on zip_master(state, city);
quit;
```

In general, you will use unique indexes on lookup tables. You can retrieve multiple records with the same non-unique key, but that is beyond the scope of this paper. Notice that you can create multiple indexes using different keys on the same data set.

The heart of the lookup is a SET statement specifying the name of the index to be used, eg:

```
      set zip_master (keep=zip city state) key=zip / unique;
```

This tells SAS to use the index to find the record that has the current value of the indexed variables. If this search succeeds, the automatic variable _iorc_ will be set to 0; otherwise it will be set to a non-zero value indicating what the problem was. In general, you need to check for only 0 or non-zero, and not the specific error that occurred.

Here is a complete lookup program. Notice the use of CALL MISSING if the lookup fails; ZIP is a retained variable because it comes from a data set, so if the lookup fails the ZIP variable will still hold the value from the last successful lookup. A unsuccessful lookup will also set the _ERROR_ variable, which you want to reset back to 0 so an error message won't be printed in the log:

```
/* The tricky thing about index lookups is that     */
/* values from the lookup table are automatically   */
/* retained, so if the lookup fails your data step  */
/* will still have the value from the previous      */
/* successful lookup.  Use CALL MISSING (or some-   */
/* thing similar) to clear it.  Because you         */
/* also have to handle the error condition set by   */
/* a failed lookup, you can also handle the value   */
/* reset then.                                      */

data idx_lkup_zip_from_state_city;

    set lookup_by_zip;

    set zip_master (keep=zip city state) key=zip / unique;

    if _iorc_ ne 0 then
        do;
        call missing(city, state);
        _error_ = 0;
        end;

run;
```

In a more complex program, other code might have set _ERROR_, and you probably want to know about that. Save the previous value of _ERROR_ before the indexed SET statement, and restore it afterwards:

```
data idx_lkup_zip_from_state_city;

    set lookup_by_zip;

    __save_error_rc = _error_;
    drop __save_error_rc;

    set zip_master (keep=zip city state) key=zip / unique;

    if _iorc_ ne 0 then
        do;
        city = '??';
        state = '??';
        _error_ = __save_error_rc;
        end;

run;
```

As with hash objects, you can add new entries to the lookup table in the middle of a data step. To do that, you must use the MODIFY statement instead of the SET statement to read the indexed data set, and that data set must be listed in the DATA statement:

```
/* The next example shows adding additional records to the  */
/* lookup table in the middle of the data step, similar to  */
/* what was done in example 06-hash.  One big difference    */
/* between this and the hash method is that the underlying  */
/* data set is updated to contain the new values.           */
/* Because you're changing the data set, you have to put    */
/* the lookup table in the DATA statement, and add the new  */
/* values with MODIFY.                                      */

data idx_lkup_zip_from_state_city_ky
     zip_master;

    /* Add new values for KY, first time only. */
    if _n_ = 1 then
        do;
        do until (nomore_adds);
            set zip_master_ky (keep=zip city state) end=nomore_adds;
            output zip_master;
        end;
        end;

    set lookup_by_state_city;

    __save_error_rc = _error_;
    drop __save_error_rc;

    zip = '??';
    modify zip_master (keep=zip city state) key=state_city / unique;

    if _iorc_ ne 0 then
        _error_ = __save_error_rc;
```

```
    output idx_lkup_zip_from_state_city_ky;

run;
```

ADDITIONAL PROGRAMS

In addition to the code shown in this paper, the sample code file includes a program that uses an indexed lookup data set to maintain a "high water mark" synthetic key, and a utility program to print all data sets in a library.

CONCLUSION

SAS software has numerous ways to perform table lookups. Review your options carefully to determine which is most appropriate for your program.

ACKNOWLEDGMENTS

I'd like to thank the Division of Research at Kaiser Permanente of Northern California for encouraging me and my colleagues to learn different ways to perform tasks in SAS software.

I'd also like to thank Mary Rosenbloom for leading me to this topic.

CONTACT INFORMATION

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

Jack Hamilton
Division of Research
Kaiser Permanente of Northern California
jack.hamilton@kp.org or jack.hamilton.kpdor@gmail.com
www.dor.kaiser.org