A SURVEY OF TABLE LOOKUP TECHNIQUES FOR THE SAS® SYSTEM

James E. Johnstone, ARC/PSG

Craig Ray, ARC/PSG

I. INTRODUCTION

This paper presents a variety of techniques to solve table lookup applications in the SAS system. Table lookup can be loosely defined as the acquisition of additional information from an auxiliary or parameter file based on the value or values of one or more variables contained in a primary file. Motivated by software engineering considerations, alternative solutions within the framework of the SAS system are presented primarily in terms of the number of operations involved in acquisition of the desired result. Sample code for each of the approaches, their relative strengths and weaknesses, and guidelines for selection are presented. Considerations of maintenance, complexity, and start-up costs are also discussed.

Performance characteristics obtained for the various alternatives on different platforms ranging from PC to mainframe are beyond the scope of this paper. The information and guidelines presented represent the combination of two related papers previously presented (references 1 and 2) with some additional speculations and considerations.

II. DEFINITIONS

There are many applications which can benefit from the use of table lookup applications. Consider a set of video retail outlets which incorporates computerized capture of business transactions with customers at time of purchase. For purposes of accounting, preparing regular customer summary statements and the like, additional demographic and other information is needed together with the transaction data. The files we will refer to in the remainder of our exploration are shown in Figure 1.

There are several terms which will be used repeatedly through the body of this discussion. Brief definitions are offered below:

Primary file - the file which is being processed, typically one record at a time, for which auxiliary information is desired (i.e., CUSTRANS).

Lookup file - a reference or parameter auxiliary file which will be visited by one or more records of the main file to obtain information (i.e., MASTCUST or MASTPROD).

Key - some variable or set of variables whose values or combination thereof are the elements in common between the primary and lookup files. Typically, key values are unique in the lookup file; this is not necessarily the case for the primary file (i.e., CUDSTID or TRANTYPE).

Lookup result - the auxiliary information obtained using the key as a reference into the lookup file.

Seek operation - scanning or searching operation involved in using the key to access the lookup file; different types of operations are thought of as conceptually equivalent (i.e boolean versus arithmetic equality).

FIGURE 1
SAMPLE DATA FOR PRIMARY AND LOOKUP FILES

PRIMARY FILE
Customer Transaction File

Customer ID (CUSTID)	Transaction Type ID (TRANTYPE)	Subtype ID (SUBID)	Date/Time Stamp (DATETIME)	Value (VALUE)	Tax (TAX)
00001	21	0000	8901151030	930.00	46.50
02173	19	1217	8901151039	22.00	1.10
02173	19	1217	8901151040	22.00	1.10
02173	02	0000	8901151041	479.00	23.95
00007	01	0000	8901151130	952.00	47.60
00007	21	0000	8901151133	1879.00	93.95
00007	13	2173	8901151133	2.50	0.13
00007	13	9718	8901151134	2.50	0.13
.	.	.	.	.	.
.	.	.	.	.	.
.	.	.	.	.	.

LOOKUP FILE 1
Master Customer File

CUSTOMER ID (CUSTID)	CUSTOMER NAME (CUSTNAME)	CUSTOMER ADDRESS (CUSTADDR)
00001	JUNE CLEAVER	911 MAPLE ST., BURBANK, CA
00007	OZZIE NELSON	123 ELM DR., APPLE VALLEY, CA
02173	TAB HUNTER	8310 SUNSET BLVD., WEST HOLLYWOOD, CA
07156	MISTER ED	1000 RODEO DR., BEVERLEY HILLS, CA
12457	JED CLAMPETT	9100 WILSHIRE BLVD., BEVERLEY HILLS
17345	DONNA REED	4300 SUNNYSLOPE VAN NUYS, CA
.	.	.
.	.	.
.	.	.
98732	M. DRYSDALE	9200 WILSHIRE BLVD, BEVERLY HILLS, CA

LOOKUP FILE 2
MASTER TRANSACTION ID FILE

TRANSACTION TYPE ID (TRANTYPE)	TRANSACTION DESCRIPTION (TRANS)
01	DIGITAL VHS VCR
02	DIGITAL BETA VCR
13	MOVIE RENTAL VHS
19	MOVIE PURCHASE BETA
21	DIGITAL MONITOR

Consider again the sample data shown in Figure 1. The primary file (CUSTRANS) consists of every customer transaction for the regional video operation. These transactions include customer ID (CUSTID), transaction type ID (TRANSID), sub transaction ID (SUBID), a date time stamp (DATETIME), cost of goods or services (VALUE), and associated taxes (TAX). A number of lookup tables can be envisioned, such as a map of transaction type IDs to descriptive labels (MASTPROD) or our master customer file (MASTCUST) which maps customer IDs to customer addresses and other individual customer information. The key to these two tables are the transaction type ID and Customer ID whose coding is arbitrary, but is presented as two and five digit values to allow for 100 and 100,000 possibilities respectively for illustrative purposes. Given these data structures, a number of questions can be posed which can benefit from table lookup mechanisms to relate information.

III. ALTERNATIVE SOLUTIONS

A variety of classical techniques are presented, some more familiar than others:

IF-THEN-ELSE statements to set the value of the lookup result;

ARRAY data structures to store the lookup table for subsequent access;

MACRO symbol tables to store the lookup table for subsequent access;

SAS SORT and MERGE to combine the primary and lookup files into a single data structure;

PROC FORMAT generation of lookup table using the PUT function to obtain the lookup result;

SAS coded binary search to perform lookup on a key ordered file;

Hashing algorithm to generate a more uniquely addressable lookup table.

Each of these approaches has advantages and disadvantages and vary in complexity. They are described in the following sections.

IV. SIMPLE APPROACHES: IF-THEN-ELSE, ARRAYS, AND MACRO SYMBOL TABLES

A typical, albeit sometimes inappropriate solution to the table lookup problem is to obtain the lookup result based on the key value in a set of if-then-else clauses as shown in Figure 2. This approach is satisfactory if there are only a few recodes in the lookup file and their values do not often change. Beyond such limited situations, this solution is a poor one. As the size of the lookup file increases, so does the average number of seek operations for each record in the primary file, which is half the number of "observations" in the lookup file. Other disadvantages include the necessity of changing the code in the DATA step which includes the if-then-else clauses; as the size of the lookup file grows, any other logic included in the DATA step becomes obscured.

In our example, the if-then-else recode solution might be appropriate for relating the value of transaction type ID, of which there are few, to descriptive information as shown, but probably not for the relation of customer IDs, of which there are many, to descriptive information from the master customer file.

FIGURE 2

SETTING TRANSACTION VALUES USING IF-THEN ELSE STATEMENTS

```
DATA REPORT;
   SET CUSTRANS;
   LENGTH TRANS $40.
   IF TRANTYPE = 1 THEN
     TRANS = 'DIGITAL VHS VCR';
   ELSE IF TRANTYPE = 2 THEN
     TRANS = 'DIGITAL BETA VCR';
   ELSE IF TRANTYPE = 13 THEN
     TRANS = 'MOVIE RENTAL VHS';
   ELSE IF TRANTYPE = 19 THEN
     TRANS = 'MOVIE PURCHASE BETA';
   ELSE IF TRANTYPE = 21 THEN
     TRANS = 'DIGITAL MONITOR';
   (subsequent processing)

RUN;
```

Two other approaches which are well suited to the former problem include arrays and macro symbol tables. The concept of arrays is presented in Figure 3. Note that the values of transaction type ID are used as indices into the array and that the values are retained for every observation in the primary file being processed. Advantages to this solution are the lack of any sort in the lookup process and a single array reference operation to obtain the result with the option of conditional lookup. Disadvantages include the amount of storage required to drag the array along with each observation of the primary file, the consecutive numbering or referencing of the array structure which may be heavily impacted by the nature of the key values, the overhead of searching the array and the problems of maintenance when the values in the lookup file change. These limitations make the solution appropriate for only smaller lookup problems limited to 40 or fewer records in the lookup file.

FIGURE 3

SETTING TRANSACTION VALUES USING AN ARRAY STRUCTURE

```
DATA REPORT;
   SET CUSTRANS;
   ARRAY TRTYP(I) $40 TRTYP1-TRTYP21;
   RETAIN TRTYP1 'DIGITAL VHS VCR'
          TRTYP2 'DIGITAL BETA VHS'
          TRTYP13 'MOVIE RENTAL VHS'
          TRTYP19 'MOVIE PURCHASE BETA'
          TRTYP21 'DIGITAL MONITOR';
   I=TRANTYPE;
   TRANS=TRTYP;
   (subsequent processing)
RUN;
```

Another solution which allows us to avoid the disadvantages cited for arrays related to excess storage, consecutive indexing, and overhead costs is the use of small macro symbol tables. The lookup table can be loaded into macro variables in a variety of ways such as reading an input file and loading variables via CALL SYMPUT. For simplicity, let's assume our table is created using %LET statements as shown in Figure 4. The lookup result in this example can be obtained by referencing the transaction key concatenated to an alpha root of the macro variables' names in a SYMGET operation. This simple approach has many advantages, among them the data driven capability of table generation, which obviates the necessity to change source code when the lookup table changes. No sorts for the lookup are required, and the lookup result may be up to 1024 characters in length, though the overhead costs of searching the macro symbol table appear to increase at a greater than linear rate as the lookup file grows in size. When the lookup file gets too large, repartitioning of space may be required to accommodate the larger table or reduce overhead.

FIGURE 4

SETTING TRANSACTION VALUES USING MACRO SYMBOL TABLES

```
%LET TRTYP1 = DIGITAL VHS VCR;
%LET TRTYP2 = DIGITAL BETA VCR;
%LET TRTYP13 = MOVIE RENTAL VHS;
%LET TRTYP19 = MOVIE PURCHASE BETA;
%LET TRTYP21 = DIGITAL MONITOR;

DATA REPORT;
   SET CUSTRANS;
   TRANS = SYMGET("TRTYP"||TRANTYPE);
   (subsequent processing)
RUN;
```

V. SAS SORT AND MERGE

Another easily coded alternative is the sort and merge of both the primary and lookup files with additional processing in the same or subsequent DATA step, depending on required sort order as shown in Figure 5. This is perhaps the simplest approach to table lookup; it can be streamlined by retaining only the required variables on input using KEEP as a DATA set option for both SAS data sets in the MERGE statement. It is assumed that the lookup table is set up as a SAS data set, as is the primary file. If the sort routine on the platform does not create a processing bottleneck, this is the desirable solution when both primary and lookup files are large. The obvious disadvantage is the required sorts. Furthermore, every record of the lookup file will be visited whether needed or not, but only once. Conditional lookup is not possible as was the case for the previous approaches; the primary file usually requires two sorts; one to put the data set in order by lookup key and another to restore the data set to its original sort order.

FIGURE 5

SETTING TRANSACTION VALUES USING SORT AND MERGE OF BOTH FILES

```
PROC SORT DATA = CUSTRANS;
  BY CUSTID;
RUN;

PROC SORT DATA = MASTCUST;
  BY CUSTID;
RUN;

DATA REPORT;
  MERGE CUSTRANS(IN=INTRANS)
        MASTCUST(KEEP=CUSTID NAME);
  IF INTRANS;
  (subsequent processing)
RUN;
```

VI. PROC FORMAT WITH PUT FUNCTION

Another option available is the expression of the lookup table as a format created by a PROC FORMAT. Subsequent access is made via a PUT function within the context of a DATA step. This approach represents the most generalized and efficient technique currently available in the SAS System and is strongly recommended when the size of the lookup file is less than approximately 10,000 values, depending on length of the result of the lookup; shorter lengths allow for 20,000 entries easily. While the use of formats for labelling values in procedure output is common, an alternative use is presented in Figure 6. Here the values of the lookup table are hard coded in a PROC FORMAT to map unique values of transaction type ID to transaction values. The TRANSVAL format is then referenced via a PUT statement for each record of CUSTRANS for which lookup is desired. Note that we again have the capability to conditionally perform lookup based on the value of some variable in the primary file. While this limited application can be easily solved using any of the other DATA step based solutions thus far presented, this technique can handle much larger tables, such as our MASTCUST file. Suppose we have approximately 5,000 entries in our master customer file; the code to access the formatted values is presented in Figure 7. Other than sort and merge, this is the only practical solution thus far presented when the lookup file becomes this large. It has the advantage of being extremely fast; SAS programs search through format tables using a balanced binary search which does not require that the lookup file be ordered by values of the key.

FIGURE 6

SETTING TRANSACTION VALUES USING SMALL FORMATS

```
PROC FORMAT;
  VALUE TRANSVAL
  1 = 'DIGITAL VHS VCR'
  2 = 'DIGITAL BETA VCR'
  13 = 'MOIVIE RENTAL VHS'
  19 = 'MOVIE PURCHASE BETA'
  21 = 'DIGITAL MONITOR';
RUN;

DATA REPORT;
   SET CUSTRANS;
   LENGTH TRANS $40.;
   TRANS = PUT(TRANTYPE,TRANSVAL.);
   (subsequent processing)
RUN;
```

FIGURE 7

SETTING CUSTOMER NAME USING A FORMAT

```
PROC FORMAT;
   VALUE $CUSTNAM;
   '00001' = 'JUNE CLEAVER'
   '00007' = 'OZZIE NELSON'
   '02173' = 'TAB HUNTER'
   '07156' = 'MISTER ED'
   '12457' = 'JED CLAMPETT'
   '17345' = 'DONNA REED'
      .          .
      .          .
      .          .
   '98732' = 'M. DRYSDALE';
RUN;

DATA REPORT;
    SET CUSTRANS;
    LENGTH NAME $40.;
    NAME = PUT(CUSTID,$CUSTNAM.);
    (subsequent processing)
RUN;
```

While we have separated the lookup file from the DATA step, we still must modify the source code module for any changes in the lookup file which presents continued constraints. A data driven alternative is presented in the macro %MAKEFMT in Figure 8, where a PROC FORMAT is generated from a lookup file stored as a SAS data set.

The code allows for external storage of the resulting format. The macro assumes that the user knows to give NAME a preceding $ symbol if it is a character format. Through the use of %STR and CALL EXECUTE functions, the format is built observation by observation from FROMDATA.

FIGURE 8

MACRO TO GENERATE PROC FORMAT FROM A SAS DATA SET

```
OPTIONS DQUOTE;

%MACRO MAKEFMT
      (FROMDATA= ,  /* INPUT SAS DATA SET */
       KEY=,        /* LOOKUP VARIABLE */
       RESULT=,     /* RESULT OF LOOKUP VARIABLE */
       OTHER=FAIL,  /* VALUE FOR OTHER */
       DDNAME=,     /* PERMANENT LIBRARY FOR FORMAT */
       NAME=);      /* NAME OF FORMAT */

%LOCAL EXTRA;
%IF %LENGTH(&DDNAME) > 0 THEN %LET EXTRA = DDNAME=&DDNAME;
%ELSE %LET EXTRA = ;

DATA _NULL_;
  IF _N_ = 1 THEN
    CALL EXECUTE ("PROC FORMAT &EXTRA;VALUE &NAME");
  IF LASTREC THEN
    %IF %LENGTH(&OTHER) > 0 %THEN
      %STR( CALL EXECUTE (" OTHER='&OTHER';RUN;") ; );
    %ELSE
      %STR( CALL EXECUTE (";RUN;") ; ) ;
  SET &FROMDATA END=LASTREC;
  %IF %SUBSTR(&NAME,1,1) = $ %THEN
    %DO; /* CHARACTER FORMAT */
CALL EXECUTE (' " '||TRIM(&KEY)||'"="'||TRIM(&RESULT||'"');
    %END; /* CHARACTER FORMAT */
  %ELSE
    %DO; /* NUMERIC FORMAT */
CALL EXECUTE (PUT(&KEY,BEST16.)||'="'||TRIM(&RESULT||'"');
    %END; /* NUMERIC FORMAT */
RUN;

%MEND MAKEFMT;
```

After the macro %MAKEFMT macro has compiled, the code to lookup the customer name for each record of the transaction file is presented in Figure 9. This minimizes the issue of maintainability since the source code need not be changed when the lookup file changes. There are additional advantages in terms of maintenance; formats can be permanently saved under most platforms requiring that %MAKEFMT only be run when the lookup file changes.

FIGURE 9

USE OF THE MACRO %MAKEFMT

```
%MAKEFMT(DATA = MASTCUST,
         KEY = CUSTID,
         RESULT = NAME,
         FORMAT = $CUSTNAM)

DATA REPORT;
     SET CUSTRANS;
     LENGTH NAME $40.;
     NAME=PUT(CUSTID,$CUSTNAM.);
     (subsequent processing)
RUN;
```

Those familiar with PROC FORMAT may have already noted the limitation of 40 characters to the lookup result. Thus, this approach might not work if we wished to obtain customer address containing street address, city and state from our MASTCUST file. We can modify the code to accommodate this requirement as shown in Figure 10. The format consists of pointers where each customer ID is associated with the observation number in the SAS data set MASTCUST. The result is obtained by an INPUT conversion of the PUT function result using this format of observation numbers; the retrieval operation is accomplished using a set statement with the POINT option. There are two significant drawbacks to this solution. First, the use of the POINT option is not possible conjointly with a BY statement in the same DATA step, eliminating the option of automated control break processing (FIRST. and LAST. variable statements). However, these can be programmatically detected through the use of temporary variables and RETAIN statements. In addition, if more than one third to one half of the observations of the lookup file are to be consulted, the increased time associated with simulated direct access can make sort and merge as attractive an alternative.

FIGURE 10

USING PROC FORMAT WITH PUT FUNCTION TO MAP LOCATION USING POINT OPTION

```
PROC FORMAT;
  VALUE $CUSTPNT;
  '00001' = '1'
  '00007' = '7'
  '02173' = '59'
  '07156' = '143'
  '12457' = '212'
  '17345' = '263'
     .      .
     .      .
     .      .
  '98732' = '5000';
RUN;

DATA REPORT;
   SET CUSTRANS;
   LENGTH ADDRESS $200.;
   PNTR = INPUT(PUT(CUSTID,$CUSTPNT.),5.);
   SET MASTCUST POINT=PNTR;
   (subsequent processing)
RUN;
```

In general, the advantages of this approach can be summarized as follows. First, the approach is extremely fast. It offers easier maintenance with lookup files created by SAS data sets and maintained separately from source code and potentially permanently saved avoiding the necessity of recreation. To implement, neither the primary file nor the lookup file need be sorted, allowing any BY group processing desired in the range of the DATA step as long as the POINT option is not required. The technique is flexible, permitting more than one lookup per observation based on different variables and different formats and allowing for conditional lookup.

There are disadvantages as well. PROC FORMAT is limited to 32,000 entries; in addition, the cost to create the format table increases with the number of entries. The rate of growth in the cost can be offset by repartitioning available memory. The cost of these disadvantages must be weighed against the frequency with which larger lookup formats must be created and the availability of permanent storage of the resulting format. For lookup files of larger than 10,000 to 20,000 observations, the sort and merge solution is preferred. If the lookup file is large and the number of visits required by the primary file is small, a binary search coded in SAS software is also an option.

VII. SAS CODED BINARY SEARCH

Binary search, a lookup technique often seen implemented in third generation languages, works on the principle of dividing an ordered table in half and comparing the obtained key value with the sought after result. After determining the nature of the relationship (which is greater in value), the ordered list is successively halved until the desired key value is found. On average, it will require INT(LOG2(N)) seek operations to find the desired key value. Thus, for example, it will require only 19 seeks of a 1,000,000 element table to obtain the result on average. This solution is clearly preferable as the size of the lookup table increases with respect to the primary file. As an alternative to the sort and merge operations, consider a primary file of size 100 and a lookup file of size of 100,000 records. The binary search will on average require less than 200 operations to obtain the lookup result; in all cases, the sort and merge requires 100,000 operations.

As coded in the SAS macro %BSEARCH presented in Figure 11, the lookup represents a portion of a DATA step and as such can be invoked via IF or %IF logic. The following points are of particular note in the use of this approach. First, the lookup file is assumed to be in a SAS data set structure sorted by the lookup key or keys. The code presented uses the SET statement with the POINT and NOBS options. The value of NOBS is determined at compile time from the descriptor portion of the SAS data set; the value is automatically retained. The value of the variable referenced in the POINT option is the observation to be read. Using the POINT option does eliminate the possibility of use of FIRST. and LAST. variables for BY group processing. As stated previously, this drawback can be overcome through the use of retained temporary variables. The macro generates a number of SAS data step variables which should be dropped by the calling DATA step for efficiency. Some of these variables are used to compute the next half point and segments to search (__FIRST, __LAST, and __MID); others are used for degenerate conditions. The variable __FOUND is initialized to zero and set to one if the search is successful. If it is still zero when the search is concluded, the key is unavailable and the result from the last unsuccessful POINT should be blanked out. The value of the variable __TOTAL is checked in value; if the value is zero, then the lookup table is empty. In this event, __FOUND is set to zero and control is returned.

FIGURE 11

SAS CODED BINARY SEARCH

```
%MACRO BSEARCH(TABLE=,      /* NAME OF LOOKUP FILE */
               KEY=,        /* KEY VARIABLE - CONCATENATED
                               FOR MULTIPLE KEY VARIABLES */
               GETVARS=,    /* RESULT OF LOOKUP */ );

/* INITIAILIZE LOCATION CONTROL */

__FOUND = 0;
__STOP = 0;
__FIRST = 0;
__LAST = __TOTAL;

/* END INITIALIZATION */

DO UNTIL (__STOP);
  __MID = INT((__FIRST + __LAST) / 2);
  SET &TABLE (RENAME = (&KEY=__TARGET) KEEP = &GETVARS &KEY)
      POINT = __MID NOBS = __TOTAL;
  IF &KEY = __TARGET THEN
    DO; /* FOUND DESIRED RECORD */
      __STOP = 1;
      __FOUND = 1;
    END; /* FOUND DESIRED RECORD */
  ELSE
    DO; /* CONTINUE SEARCH */
      IF &KEY < __TARGET THEN
        DO; /* SEARCH LOWER SUBLIST */
          __LAST = __MID - 1;
        END; /* SEARCH LOWER SUBLIST */
      ELSE
        DO; /* SEARCH UPPER SUBLIST */
          __FIRST = __MID + 1;
        END; /* SEARCH UPPER SUBLIST */
      IF __FIRST > __LAST THEN
        DO; /* RECORD NOT ON FILE */
          __STOP = 1;
        END; /* CONTINUE SEARCHING */
    END; /* CONTINUE SEARCH */
%MEND BSEARCH;
```

The code generated by the macro %BSEARCH can be invoked conditionally or unconditionally as shown in Figure 12. As an illustration of the method's flexibility, %BSEARCH can be called multiple times either for the same or multiple lookup files in the same execution of the calling DATA step. In summary, the other advantages of the SAS coded binary search include the following: no sort requirement for the primary file; each lookup requires on average only INT(LOG2(N)) seek operations; relative freedom from maintenance concerns with respect to source code; and freedom from the 32K element limit imposed by PROC FORMAT. Its disadvantages include a required sort of the lookup file by the key and the somewhat inefficient processing time of SET with the POINT option. On many platforms, the POINT option to simulate direct access read can take as long as three times that of a sequential read operation. While the techniques presented thus far should solve most problems, a more powerful technique is hashing.

FIGURE 12
USE OF %BSEARCH MACRO

```
DATA REPORT(DROP = ..............);
   SET CUSTRANS;
   IF PUT (TRANTYPE,TRANSVAL.) =
     'DIGITAL VHS VCR' THEN
       DO; /* INVOKE BINARY SEARCH */
         %BSEARCH(TABLE = MASTCUST,
                  KEY = CUSTID,
                  GETVARS = NAME);
       END; /* INVOKE BINARY SEARCH */
   (subsequent processing)
RUN;
```

VIII. IMPLEMENTATION OF HASHING IN THE SAS SYSTEM

Hashing represents perhaps the most powerful and fastest generalized table lookup technique. While its concepts and coding are in many ways the most esoteric of the solutions presented, its power as an alternative, particularly to SAS sort and merge, make it worthy of consideration. The distinction to grasp here is the rearrangement of observations in the lookup file such that the value of the key indicates the placement of the observation in the hash table. If done efficiently, which generally requires at least twice the space of the lookup file in most cases, the lookup result is obtained on average in approximately 1.5 seek operations.

The creation of the hash table is accomplished by performing some operation on the key variable of the lookup file which yields the address of the key in the hash table. To perform a lookup for an observation in the primary file, the same operation is performed on the key to obtain the address. As an example, our MASTPROD lookup file could be hashed into a 21 element table and lookup is performed by using the transaction type ID as the POINT= variable. While only one operation per primary file observation is required, the hash table is much larger than our original structure. Consider our MASTCUST file; if it had 5,000 observations but 100,000 possible key values the resulting hash table using this approach would be large indeed. Cost/availability of storage space for the hash table and the associated costs of maintenance together with the difficulty of selecting a good hash algorithm are the biggest considerations in the evaluation of suitability for this solution.

Our first hash algorithm provides a unique mapping and requires only one seek operation, but results in very large tables. Typically, hash algorithms using a MOD function with a prime number base seem to operate well. The difficulty is the selection of the appropriate base. Once an algorithm is chosen which hashes the lookup file into an arrangement smaller than all possible key values, the operation may hash two or more of the lookup records to the same address in the hash table. These are referred to as collisions. Collisions are handled by loading the first observation that hashes to a given address to the hash table and loading collisions sequentially as they occur into an overflow table. Pointers to the first and last occurrences in the overflow table are maintained in the hash table itself. Thus, when the hash table is referenced to perform a lookup based on the imposition of the hash algorithm on the key value in the primary file and the desired result is not obtained, the overflow table is sequentially searched using the POINT option ranging from the first to last collisions until the keys are equal in value.

Sample code to implement the hashing solution is presented in Figures 13 and 14. Figure 13 shows the creation HASHVAR as the rearrangement of the lookup table with an arbitrary prime base. The table is sorted by the resulting ADDRESS; the HASHTBLE and OVERFLOW SAS data set structures are created as shown. In Figure 14, the code to search the hash and overflow tables is presented. While this approach represents the fastest generalized lookup technique available currently within the SAS system, the costs of creating and storing the hash and overflow tables must be taken into consideration; this alternative is generally only recommended as an alternative to the SAS sort and merge approach.

FIGURE 13

SAMPLE HASHING ALGORITHM AND CREATION OF HASH TABLE

```
%MACRO HASHTBLE(PRIME=, ); /* PRIME NUMBER CHOSEN FOR MOD
                              FUNCTION */

  /* THIS MACRO OPERATES ON THE LOOKUP SAS DATA SET TO    */
  /* CREATE A SORTED HASHED RESULT WHICH THEN IS USED TO */
  /* CREATE THE HASH AND OVERFLOW TABLES FROM WHICH      */
  /* LOOKUPS WILL THEN BE MADE.                          */
  /*                                                     */
  /* DATA SET CONTENTS                                   */
  /*       HASHVAR - KEY, ADDRESS, RESULT OF LOOKUP      */
  /*       HASHTBLE - TBLE_KEY, ADDRESS, FIRST AND LAST  */
  /*                  POINTERS TO OVERFLOW, RESULT OF    */
  /*                  LOOKUP                             */
  /*       OVERFLOW - TBLE_KEY, ADDRESS, RESULT OF LOOKUP */

DATA HASHVAR;
  SET LOOKUP;
  ADDRESS = MOD(KEY,&PRIME);
RUN;

PROC SORT DATA=HASHVAR;
  BY ADDRESS;
RUN;

DATA HASHTBLE OVERFLOW(DROP = FIRST LAST);
  LENGTH TBLE_KEY $ 11;
  DROP OVEROBS HASHOBS KEY;
  RETAIN FIRST;
  SET HASHVAR;
  BY ADDRESS
  TBLE_KEY = ' ';
  DO WHILE(ADDRESS > HASHOBS + 1);
    OUTPUT HASHTBLE;
    HASHOBS + 1;
  END;
  TBLE_KEY = KEY;
  IF FIRST.ADDRESS AND LAST.ADDRESS THEN
    DO; /* SINGLE MAP - NO OVERFLOW */
      OUTPUT HASHTBLE;
      HASHOBS + 1;
    END; /* SINGLE MAP - NO OVERFLOW */
  ELSE IF FIRST.ADDRESS THEN
    DO; /* SEND TO OVERFLOW AND INITIALIZE FIRST */
      OVEROBS + 1;
      FIRST = OVEROBS;
      OUTPUT OVERFLOW;
    END; /* SEND TO OVERFLOW AND INITIALIZE FIRST */
  ELSE IF LAST.ADDRESS THEN
    DO; /* OUTPUT TO HASHTBLE */
      LAST = OVEROBS;
      OUTPUT HASHTBLE;
      HASHOBS + 1;
      FIRST = .;
    END; /* OUTPUT TO HASHTBLE */
  ELSE
    DO; /* OUTPUT TO OVERFLOW */
      OVEROBS + 1;
      OUTPUT OVERFLOW;
    END; /* OUTPUT TO OVERFLOW */
RUN;

%MEND HASHTBLE;
```

FIGURE 14

USE OF THE HASH TABLE FOR LOOKUP

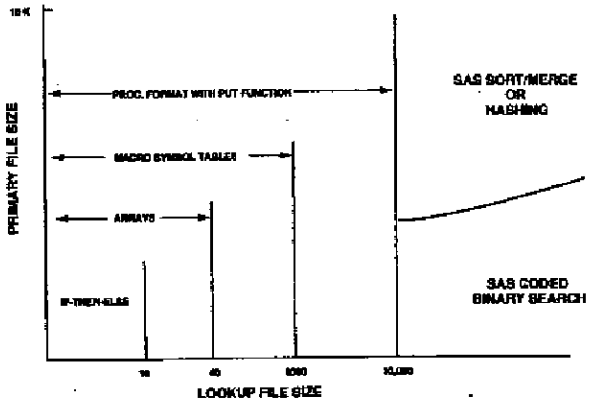
```
%MACRO HASHFIND(PRIME = ,); /* PRIME SELECTED */

  /* THIS MACRO PERFORMS THE ACTUAL LOOKUP. IT ASSUMES     */
  /* THAT THE VARIABLE NAMED ADDRESS HAS BEEN CREATED      */
  /* CONTAINING THE OBSERVATION TO BE REFERENCED IN SAS    */
  /* DATA SET HASHTBLE. THE MACRO MAY THEN GO TO SAS DATA  */
  /* SET OVERFLOW BASED ON POINTERS IN HASHTBLE            */

DATA FIND;
  SET MAIN;
  ADDRESS = MOD(KEY,&PRIME);
  IF ADDRESS <= THASHOBS THEN
    DO; /* SEARCH HASHTBLE AND/OR OVERFLOW */
      SET HASHTBLE POINT = ADDRESS NOBS = THASHOBS;
      IF KEY NE TBLE_KEY AND TBLE_KEY NE ' ' AND LAST
        NE . THEN DO; /* SEARCH OVERFLOW */
          OVERPTR = FIRST;
          DO UNTIL (OVERPTR > LAST OR KEY = TBLE_KEY);
            IF OVERPTR <= TOVEROBS THEN
              DO; /* PERFORM SETS */
                SET OVERFLOW POINT = OVERPTR NOBS = TOVEROBS;
                OVERPTR + 1;
              END; /* PERFORM SETS */
              ELSE OVERPTR = LAST + 1;
          END;
        END; /* SEARCH OVERFLOW */
    END; /* SEARCH HASTBLE AND/OR OVERFLOW */
RUN;

%MEND HASHFIND;
```

IX. GENERAL RECOMMENDATIONS

Each of the alternatives presented has advantages and disadvantages which have been described in previous sections. Eliminating the issue of different platforms and overlooking maintenance concerns for the most part, general suggestions for which alternative to use are shown in Figure 15.

FIGURE 15

GENERAL RECOMMENDATIONS FOR METHOD SELECTION



X. FUTURE DIRECTIONS - INDEXING SAS DATA SETS

The Version 6 release intends to provide an index data structure which is associated with each SAS data set. Based on the values of one or more variables, a data set will be able to have single or composite indexes. These index data structures will improve processing time in the use of WHERE clauses and provide an access method which will be a heavy contender with the methods presented in this paper. The availability of this data structure and how it can be used within the context of the SAS system are not yet clear. Once implemented, further investigation on the part of users will be required to determine where this access method falls in relation to the others herein presented.

XI. ACKNOWLEDGEMENTS

The authors wish to express their appreciation for all the past work on the part of other researchers whose ideas have presented a wealth of information from which to work. In particular, this year's presentation would not have been possible without the help of the following individuals: Ian Whitlock, who provided improved %MAKEFMT code and directed editing efforts of our review team; Neil Howard, who made sure I kept on track as the conference approached; Warren Repole, who provided a great deal of coaching in the word processing and font presentation of the document; and Paige Armstrong, whose last minute editing and production assistance was invaluable.

XII. REFERENCES

Interested readers should consult the original sources referenced for more detailed background.

Ray, Craig, 'A Comparison of Table Lookup Techniques', SUGI '86.

Ray, Craig, 'Implementation of a Hashing Routine in SAS Software', SUGI '87.

AUTHOR CONTACT

The authors may be contacted at:

601 Indiana Avenue
Suite 1000
Washington, D.C. 20004

(202) 737-2666