Paper 11-27

Table Lookup: Techniques Beyond the Obvious

Nancy Croonen, CC Training Services, Belgium
ir. Henri Theuwissen, SOLID Partners, Belgium

ABSTRACT

Table lookup operations are often the most time consuming part of many SAS® programs. In this paper we will combine two SAS data sets by using the value of a specific variable to locate information in an auxiliary or lookup SAS data set and add it to information from the primary SAS data set. Base SAS software offers a broad range of techniques to perform table lookup operations. Do you use the most obvious technique or do you evaluate multiple techniques and determine which technique is the most efficient way to perform the lookup?

INTRODUCTION

This paper discusses seven different approaches to perform the table lookup in terms of processing time and the complexity of coding. The following programming techniques, ranging from fairly straightforward to more complicated but also from less to more efficient, are discussed using examples:

1. DATA step MERGE statement
2. SQL inner join
3. SQL subquery
4. FORMAT procedure and PUT function
5. SET statements and KEY = option
6. CALL EXECUTE routine
7. SQL INTO clause

Benchmarking results are summarized with graphs and tables. The paper addresses base SAS and is intended for intermediate users of SAS.

TERMINOLOGY

The following terms are frequently used throughout the paper:

- The **primary file** is the file for which you want to obtain auxiliary information.
- The **lookup file** is an auxiliary file that is maintained separately from the primary file and that is referenced for one or more of the observations of the primary file.
- The **key variable** is the variable or variables whose values are the common elements between the primary file and the lookup file. Typically, key values are unique in the lookup file but not necessarily unique in the primary file.
- The **lookup result** is the auxiliary information obtained using the key variable or variables as reference into the lookup file.

INPUT SAS DATA SETS

The examples in this paper use the following input SAS data sets containing fictitious data:

- The lookup SAS data set SUGI27.COMPANY contains address information of all Belgian companies. The SAS data set contains 273.709 observations and the following variables:
  - VAT_NUMBER
  - COMPANY_NAME
  - STREET
  - POSTAL_CODE
  - CITY

  The key variable VAT_NUMBER uniquely identifies a company. Each VAT number only appears once in the input SAS data set SUGI27.COMPANY.
- The SAS data set SUGI27.COMPANY_INDEXED is a copy of the SAS data set SUGI27.COMPANY, but contains a unique index on the variable VAT_NUMBER. The index is a separate structure that contains the data values of the variable VAT_NUMBER paired with a location identifier for the observations containing the value.
- The primary SAS data set SUGI27.BAD_DEBTOR contains a list of companies that are registered as bad debtors. The SAS data set contains 200 observations and contains only the key variable VAT_NUMBER. Each VAT number only appears once in the SAS data set SUGI27.BAD_DEBTOR.
- The SAS data set SUGI27.SALES contains sales information in the year 2001 of all Belgian companies. The SAS data set contains 1.506.206 observations and the following variables:
  - VAT_NUMBER
  - DATE
  - SALES

  The combination of the variables VAT_NUMBER and DATE uniquely identifies a transaction. Each VAT number can appear multiple times in the SAS data set SUGI27.SALES.

  The SAS data set SUGI27.SALES contains an index on the variable VAT_NUMBER.

Observations in SUGI27.BAD_DEBTOR and SUGI27.COMPANY or SUGI27.COMPANY_INDEXED are related by common values for VAT_NUMBER. There is a one-to-one relationship between the primary SAS data set SUGI27.BAD_DEBTOR and the lookup SAS data set SUGI27.COMPANY or the indexed lookup SAS data set SUGI27.COMPANY_INDEXED which implies that each value of the variable VAT_NUMBER occurs no more than once in each SAS data set. In other words, a single observation in the SAS data set SUGI27.BAD_DEBTOR is related to a single observation in the SAS data set SUGI27.COMPANY or the SAS data set SUGI27.COMPANY_INDEXED. We will use the value of the key variable VAT_NUMBER from the primary SAS data set SUGI27.BAD_DEBTOR to locate the associated address information in the lookup SAS data set SUGI27.COMPANY or SUGI27.COMPANY_INDEXED and add it to information from the

primary SAS data set.

Observations in the SAS data sets SUGI27.BAD_DEBTOR and SUGI27.SALES are also related by common values for VAT_NUMBER. There is a one-to-many relationship between SUGI27.BAD_DEBTOR and SUGI27.SALES which implies that each value of the variable VAT_NUMBER occurs no more than once in the SAS data set SUGI27.BAD_DEBTOR but may occur more than once in the SAS data set SUGI27.SALES. In other words, a single observation in BAD_DEBTOR may be related to multiple observations in SUGI27.SALES. We will combine the two SAS data sets by using the value of the key variable VAT_NUMBER from the SAS data set SUGI27.BAD_DEBTOR to locate the associated sales information in the SAS data set SUGI27.SALES.

SETTING SAS SYSTEM OPTIONS

Before getting started, we will turn on some SAS system options to make as much information as possible available about the execution of the SAS programs:

- The **FULLSTIMER** system option writes all the system performance statistics to the LOG window. The SAS System writes to the LOG a complete list of computer resources used for each step and the entire SAS session. The type of statistics written varies with host systems. Computer resources can be measured in the following terms:
  - CPU time is the actual time spent on a task. The CPU time is the amount of time the Central Processing Unit uses to perform the requested tasks, including calculations, reading and writing data, ...
  - I/O is a measurement of the Input (read) and Output (write) operations as data and programs are moved from a storage device to memory (input) or from memory to a storage or display device (output).
  - Memory is the size of the work area that the CPU requires to hold the executable program modules, data, and buffers.
- The **MSGLEVEL =** system option controls the level of detail in messages that are written to the LOG window. MSGLEVEL = N prints notes, warnings, and error messages only. This is the default. MSGLEVEL = I prints additional notes pertaining to index usage, merge processing, and sort utilities along with standard notes, warnings, and error messages.
- The **SYMBOLGEN** system option writes the results of resolving macro variable references to the LOG window.

```
OPTIONS FULLSTIMER MSGLEVEL = I SYMBOLGEN;
```

DATA STEP MERGE STATEMENT

The first technique, which is very easy to code, uses the MERGE and BY statements in a DATA step. The observations from the SAS data sets SUGI27.BAD_DEBTOR and SUGI27.COMPANY or SUGI27.COMPANY_INDEXED are combined into a single observation in the new SAS data set according to the values of the common variable VAT_NUMBER. Before you can perform a match-merge, both SAS data sets must be sorted by the variable VAT_NUMBER or they must have an index on the variable VAT_NUMBER.

In the following example, both input SAS data sets are not arranged in order of the values of the variable VAT_NUMBER and they also have no index on the variable VAT_NUMBER. So before match-merging the two SAS data sets in a DATA step, we must sort the SAS data sets.

PROGRAM 1-A

```
PROC SORT DATA = SUGI27.COMPANY
          OUT  = COMPANY;
   BY VAT_NUMBER;
RUN;

PROC SORT DATA = SUGI27.BAD_DEBTOR
          OUT  = BAD_DEBTOR;
   BY VAT_NUMBER;
RUN;

DATA MERGE_SORTED;
   MERGE BAD_DEBTOR (IN = BD)
         COMPANY;
   BY VAT_NUMBER;
   IF BD;
RUN;
```

The IN = data set option creates and names a variable BD that indicates whether the input SAS data set BAD_DEBTOR contributed data to the current output observation. BD is a temporary numeric variable with values of 0 or 1. The value 0 indicates that the input SAS data set BAD_DEBTOR did not contribute to the current output observation whereas the value 1 indicates that the input SAS data set BAD_DEBTOR contributed to the current output observation. The variable BD is available to programming statements during the DATA step, but is not included in the SAS data set being created.

The subsetting IF statement causes the DATA step to continue processing only those observations that meet the condition of the expression specified in the IF statement. Therefore, the resulting SAS data set contains only observations to which the input SAS data set BAD_DEBTOR contributed.

Because no index exists on the variable VAT_NUMBER in neither of both input SAS data sets, the observations are read sequentially in the order in which they appear in the input SAS data sets.

In the following example, SUGI27.COMPANY_INDEXED contains an index on the variable VAT_NUMBER. Before match-merging the two SAS data sets in a DATA step, we must only sort the input SAS data set SUGI27.BAD_DEBTOR.

PROGRAM 1-B

```
PROC SORT DATA = SUGI27.BAD_DEBTOR
          OUT  = BAD_DEBTOR;
   BY VAT_NUMBER;
RUN;

DATA MERGE_INDEXED;
   MERGE BAD_DEBTOR (IN = BD)
         SUGI27.COMPANY_INDEXED;
   BY VAT_NUMBER;
   IF BD;
RUN;
```

SQL INNER JOIN

The second technique, which is also fairly easy to code and understand, uses an SQL inner join. If you are familiar with SQL (Structured Query Language), you may want to use PROC SQL instead of the DATA step.

PROGRAM 2-A

```
PROC SQL;
   CREATE TABLE JOIN_SORTED AS
      SELECT BD.*
         FROM BAD_DEBTOR BD,
              COMPANY C
         WHERE BD.VAT_NUMBER = C.VAT_NUMBER;
QUIT;
```

Conceptually, a query with a join and a WHERE expression is evaluated in two phases. First the FROM clause is processed. SQL internally builds a virtual, temporary join table by combining each row from BAD_DEBTOR with every row from COMPANY. The result of this combination is the Cartesian product of the two tables. Next, the WHERE expression is processed. Only rows that satisfy the WHERE clause condition are selected from this join table.

While it is helpful to imagine that SQL builds a temporary, internal join table for every join, this is often not the case. In reality, the SQL procedure optimizer breaks the Cartesian product into smaller pieces. SAS data sets are stored in pages that contain a certain number of observations. To reduce I/O, the SQL procedure optimizer makes use of these pages in its processing. During a two-way join, the following tasks are completed:

1. The first page from the first table is read into memory, together with as many of the first pages from the second table as will fit into available memory.
2. Valid rows are selected.
3. The first page of the first table is kept in memory. As many subsequent pages from the second table that will fit into memory are read and step 2 is repeated.
4. All pages from the second table are processed in combination with page 1 from the first table. Steps 1 through 4 are then repeated for page 2 from the first table. The entire process stops once all rows in both tables are processed.

The SQL procedure optimizer can process an equi-join (a join on an equals condition) even more efficiently. During a two-way equi-join, the following tasks are completed:

1. Both tables are sorted by the matching column (if necessary) and are grouped by the matching column's value into chunks.
2. The Cartesian product is only performed on matching chunks of data.
3. Once a chunk of data is processed, it is not processed again.

In the previous example, both tables were already sorted by VAT_NUMBER. The next example will illustrate that this is not a prerequisite for an SQL join.

PROGRAM 2-B

```
PROC SQL;
   CREATE TABLE JOIN_UNSORTED AS
      SELECT BD.*
         FROM SUGI27.BAD_DEBTOR BD,
              SUGI27.COMPANY C
         WHERE BD.VAT_NUMBER = C.VAT_NUMBER;
QUIT;
```

An index can improve the processing of an SQL join. In the following example, the column VAT_NUMBER that participates in the join is indexed in the table SUGI27.COMPANY_INDEXED.

PROGRAM 2-C

```
PROC SQL;
   CREATE TABLE JOIN_INDEXED AS
      SELECT BD.*
         FROM SUGI27.BAD_DEBTOR BD,
              SUGI27.COMPANY_INDEXED C
         WHERE BD.VAT_NUMBER = C.VAT_NUMBER;
QUIT;
```

SQL SUBQUERY

The third technique uses an SQL subquery. Often an SQL join can also be expressed as a subquery. While an SQL join combines multiple tables into a new table, an SQL subquery (enclosed in parentheses) selects rows from one table based on values in another table. A subquery, or inner query, is a query-expression that is nested as part of another query-expression. Depending on the clause that contains it, a subquery can return a single value or multiple values. The IN condition is used to include a subquery that returns multiple values. Subqueries are most often used in the WHERE and the HAVING expressions.

The following example uses a subquery to connect the VAT numbers from the SUGI27.BAD_DEBTOR table with their address information in the SUGI27.COMPANY table. VAT numbers appear in both tables and therefore link the tables. The subquery is evaluated first and builds a virtual, internal table consisting of the VAT numbers from the SUGI27.BAD_DEBTOR table. These VAT numbers become the set of values for the IN condition in the WHERE expression of the outer query and are used to select rows from the SUGI27.COMPANY table.

PROGRAM 3-A

```
PROC SQL;
   CREATE TABLE SUBQUERY_UNSORTED AS
      SELECT *
         FROM SUGI27.COMPANY
         WHERE VAT_NUMBER IN
            (SELECT VAT_NUMBER
                FROM SUGI27.BAD_DEBTOR);
QUIT;
```

An index may also improve the processing of an SQL subquery. In the following example, the column VAT_NUMBER in the WHERE expression of the outer query is indexed.

PROGRAM 3-B

```
PROC SQL;
   CREATE TABLE SUBQUERY_INDEXED AS
      SELECT *
         FROM SUGI27.COMPANY_INDEXED
         WHERE VAT_NUMBER IN
            (SELECT VAT_NUMBER
                FROM SUGI27.BAD_DEBTOR);
QUIT;
```

FORMAT PROCEDURE AND PUT FUNCTION

The fourth technique, which is a little harder to code and understand, relates data using a user-written format. Dynamically build the format and retrieve the formatted values. This technique is efficient when you have a large lookup SAS data set whose retrieved values remain fairly constant.

First, create the input control data set FMTIN that you can use to pass information from the SAS data set SUGI27.BAD_DEBTOR to the FORMAT procedure to dynamically build the format. The input control data set should at least contain the variables FMTNAME, START, and LABEL. Define the format $VAT_FMT, which associates the character string 'BAD DEBTOR' with all the VAT numbers listed.

PROGRAM 4-A

```
DATA FMTIN;
   SET SUGI27.BAD_DEBTOR
       (RENAME = (VAT_NUMBER = START));
   RETAIN LABEL   'BAD DEBTOR'
          FMTNAME '$VAT_FMT';
RUN;
```

The input control data set assumes that the ending value of the format range is equal to the value of START if no variable named END is found. The input control data set does not require the remaining variables created for an output control data set.

Specify the FMTIN data set in the CNTLIN = option as input to the FORMAT procedure. PROC FORMAT uses the data in the input control data set to dynamically build the format.

```
PROC FORMAT LIB = SUGI27
            CNTLIN = FMTIN
            FMTLIB;
RUN;
```

The LIB = option in the PROC FORMAT statement specifies the data library, and optionally, the catalog in which to store the formats. Without the LIB = option, formats are stored in the WORK.FORMATS catalog and only exist for the duration of the SAS session.

The FMTLIB option prints information about all the formats and informats in the catalog that is specified in the LIB = option. The printed output is a table for each format or informat entry in the catalog. The output also contains global information and the specifics of each range and value (LABEL) pair defined for the format or informat. To get information only about specific formats or informats, subset the catalog using the SELECT or EXCLUDE statement.

By default, only the WORK.FORMATS catalog is searched for user-written formats. Set the FMTSEARCH = system option to identify the catalogs to be searched for formats. The WORK.FORMATS catalog is always searched first, unless it appears in the FMTSEARCH = list. The LIBRARY.FORMATS catalog is searched after WORK.FORMATS and before anything else in the FMTSEARCH = list, unless it appears in the FMTSEARCH = list. Catalogs in the list are searched in the order in which they appear in the list. If only the libref is given, SAS assumes that FORMATS is the catalog name.

```
OPTIONS FMTSEARCH = (SUGI27 WORK LIBRARY);
```

Finally, to create a subset of the lookup SAS data set SUGI27.COMPANY based on the formatted values, use the user-written format $VAT_FMT within a subsetting IF statement. Apply the user-written format $VAT_FMT to the data values of the variable VAT_NUMBER by using the PUT function. Here, the subsetting IF statement causes the DATA step to continue processing only those observations for which the formatted value of VAT_NUMBER is BAD DEBTOR. If the expression is false, the current observation is not written to the output SAS data set.

```
DATA FORMAT_DATA_IF;
   SET SUGI27.COMPANY;
   IF PUT (VAT_NUMBER, $VAT_FMT.)
      = 'BAD DEBTOR';
RUN;
```

Repeat the previous example but use a WHERE statement instead of a subsetting IF statement. Using the WHERE statement may improve the efficiency of your SAS program because the SAS System is not required to read all observations from the input SAS data set SUGI27.COMPANY. The WHERE statement selects observations before they are brought into the program data vector. The subsetting IF statement selects observations after they have been read into the program data vector.

PROGRAM 4-B

```
DATA FMTIN;
   SET SUGI27.BAD_DEBTOR
       (RENAME = (VAT_NUMBER = START));
   RETAIN LABEL   'BAD DEBTOR'
          FMTNAME '$VAT_FMT';
RUN;

PROC FORMAT LIB = SUGI27
            CNTLIN = FMTIN
            FMTLIB;
RUN;

OPTIONS FMTSEARCH = (SUGI27 WORK LIBRARY);
```

```
DATA FORMAT_DATA_WHERE;
   SET SUGI27.COMPANY;
   WHERE PUT (VAT_NUMBER, $VAT_FMT.)
         = 'BAD DEBTOR';
RUN;
```

You could also create a subset of the lookup SAS data set SUGI27.COMPANY based on the formatted values by using the PUT function in the WHERE clause of an SQL query.

PROGRAM 4-C

```
DATA FMTIN;
   SET SUGI27.BAD_DEBTOR
       (RENAME = (VAT_NUMBER = START));
   RETAIN LABEL   'BAD DEBTOR'
          FMTNAME '$VAT_FMT';
RUN;

PROC FORMAT LIB = SUGI27
            CNTLIN = FMTIN
            FMTLIB;
RUN;

OPTIONS FMTSEARCH = (SUGI27 WORK LIBRARY);

PROC SQL;
   CREATE TABLE FORMAT_SQL AS
      SELECT *
         FROM SUGI27.COMPANY
         WHERE PUT (VAT_NUMBER, $VAT_FMT.)
               = 'BAD DEBTOR';
QUIT;
```

Repeat PROGRAM 4-B but use the indexed SAS data set SUGI27.COMPANY_INDEXED instead of SUGI27.COMPANY. The LOG window will not display any messages that the index VAT_NUMBER was selected for WHERE clause optimization. The index VAT_NUMBER is not considered for use because the PUT function appears in the WHERE expression.

PROGRAM 4-D

```
DATA FMTIN;
   SET SUGI27.BAD_DEBTOR
       (RENAME = (VAT_NUMBER = START));
   RETAIN LABEL   'BAD DEBTOR'
          FMTNAME '$VAT_FMT';
RUN;

PROC FORMAT LIB = SUGI27
            CNTLIN = FMTIN
            FMTLIB;
RUN;

OPTIONS FMTSEARCH = (SUGI27 WORK LIBRARY);

DATA FORMAT_DATA_WHERE_INDEXED;
   SET SUGI27.COMPANY_INDEXED;
   WHERE PUT (VAT_NUMBER, $VAT_FMT.)
         = 'BAD DEBTOR';
RUN;
```

Also repeat PROGRAM 4-C but replace the SAS data set SUGI27.COMPANY with the indexed SAS data set SUGI27.COMPANY_INDEXED. Again the index VAT_NUMBER is not considered for use because the PUT function appears in the WHERE expression.

PROGRAM 4-E

```
DATA FMTIN;
   SET SUGI27.BAD_DEBTOR
       (RENAME = (VAT_NUMBER = START));
   RETAIN LABEL   'BAD DEBTOR'
          FMTNAME '$VAT_FMT';
RUN;

PROC FORMAT LIB = SUGI27
            CNTLIN = FMTIN
            FMTLIB;
RUN;

OPTIONS FMTSEARCH = (SUGI27 WORK LIBRARY);

PROC SQL;
   CREATE TABLE FORMAT_SQL_INDEXED AS
      SELECT *
         FROM SUGI27.COMPANY_INDEXED
         WHERE PUT (VAT_NUMBER, $VAT_FMT.)
               = 'BAD DEBTOR';
QUIT;
```

SET STATEMENTS AND KEY = OPTION

The fifth technique, which is more complicated to understand, uses an index to perform the table lookup. This technique is especially appropriate when the lookup SAS data set is large and has an index.

The first SET statement reads observations from the primary SAS data set SUGI27.BAD_DEBTOR sequentially. The second SET statement with the KEY = option reads observations from the lookup SAS data set SUGI27.COMPANY_INDEXED directly based on the value of the key variable VAT_NUMBER. In other words, the index VAT_NUMBER is used to locate observations in the lookup SAS data set SUGI27.COMPANY_INDEXED that have key values equal to the current value of the key variable VAT_NUMBER supplied by the primary SAS data set SUGI27.BAD_DEBTOR. Observations are written to the output SAS data set only when a match occurs in the lookup SAS data set.

PROGRAM 5-A

```
DATA SET_SET_KEY;
   SET SUGI27.BAD_DEBTOR;
   SET SUGI27.COMPANY_INDEXED
       KEY = VAT_NUMBER;
RUN;
```

This program works as expected only if the indexed SAS data set contains at most one observation with the same value for the key variable. The SAS data set SUGI27.SALES contains an index on the variable VAT_NUMBER. However, there is a one-to-many relationship between the SAS data sets SUGI27.BAD_DEBTOR and SUGI27.SALES. This means that a single observation in BAD_DEBTOR may be related to multiple observations in SUGI27.SALES. Use the same technique as in the previous example to combine the SAS data set SUGI27.BAD_DEBTOR and the indexed SAS data set SUGI27.SALES.

```
DATA SALES_SET_SET_KEY;
   SET SUGI27.BAD_DEBTOR;
   SET SUGI27.SALES KEY = VAT_NUMBER;
RUN;
```

Although you do not receive any error messages, the program above will not produce the desired result. The first SET statement reads observations from SUGI27.BAD_DEBTOR sequentially. The second SET statement with the KEY = option locates and reads only the first observation in SUGI27.SALES that has the key value equal to the current value of the key variable VAT_NUMBER. However, all other techniques discussed in this paper could be used to combine the SAS data sets SUGI27.BAD_DEBTOR and SUGI27.SALES in a correct way. For example, use an SQL inner join to obtain the desired result.

```
PROC SQL;
   CREATE TABLE SALES_JOIN AS
      SELECT S.*
         FROM SUGI27.BAD_DEBTOR BD,
              SUGI27.SALES S
         WHERE BD.VAT_NUMBER = S.VAT_NUMBER;
QUIT;
```

CALL EXECUTE ROUTINE

The sixth technique, which is less obvious but often very efficient, uses the CALL EXECUTE routine to create dynamic SAS code in a DATA step. The list of VAT numbers in the primary SAS data set SUGI27.BAD_DEBTOR is used to dynamically generate the WHERE statement to subset the lookup SAS data set SUGI27.COMPANY. CALL EXECUTE adds the generated SAS statement(s) to the input stack. The SAS statement(s) will execute immediately after the end of the DATA step containing the CALL EXECUTE routine.

During the first iteration of the DATA step, the following SAS code should be generated and added to the input stack:

```
DATA CALL_EXECUTE;
   SET SUGI27.COMPANY;
   WHERE VAT_NUMBER IN (
```

During each iteration of the DATA step, an observation of the primary SAS data set SUGI27.BAD_DEBTOR is read. Each time the current value of the variable VAT_NUMBER should be enclosed in quotes and also added to the input stack:

```
"426.126.740"
"454.010.874"
"444.925.439"
      …
"459.650.534"
```

During the last iteration of the DATA step, the following SAS code should be generated and added to the input stack:

```
                                    );
RUN;
```

Use the CALL EXECUTE routine to create the WHERE clause dynamically in the DATA step.

PROGRAM 6-A

```
DATA _NULL_;
   SET SUGI27.BAD_DEBTOR END = EOF;
   IF _N_ = 1 THEN DO;
      CALL EXECUTE ('DATA CALL_EXECUTE;');
      CALL EXECUTE ('SET SUGI27.COMPANY;');
      CALL EXECUTE ('WHERE VAT_NUMBER IN (');
   END;
   CALL EXECUTE (QUOTE (VAT_NUMBER));
   IF EOF THEN DO;
      CALL EXECUTE (');');
      CALL EXECUTE ('RUN;');
   END;
RUN;
```

The END = data set option is used in the SET statement to create and name a variable EOF whose value is used to detect the end-of-file. EOF is a temporary numeric variable with values of 0 or 1. The value 0 indicates the SET statement did not read the last observation in the SAS data set SUGI27.BAD_DEBTOR yet whereas the value 1 indicates the SET statement read the last observation in the SAS data set. The variable EOF is available anywhere in the DATA step, but is not added to the new SAS data set being created.

The automatic variable _N_ is created for each DATA step. The variable _N_ is initially set to 1. Each time the DATA step loops past the DATA statement, the variable _N_ is incremented by 1. So the value of _N_ represents the number of times the DATA step has iterated.

The CALL EXECUTE routine specifies a character expression that yields one or more SAS statements. The character expression can consist of constant text enclosed in single or double quotes, SAS variables, and SAS expressions. The QUOTE function adds double quotation marks to the current value of the variable VAT_NUMBER.

Repeat the previous example but use the indexed SAS data set SUGI27.COMPANY_INDEXED instead of SUGI27.COMPANY. Verify in the LOG window whether the index VAT_NUMBER was selected for WHERE clause optimization.

PROGRAM 6-B

```
DATA _NULL_;
   SET SUGI27.BAD_DEBTOR END = EOF;
   IF _N_ = 1 THEN DO;
      CALL EXECUTE
         ('DATA CALL_EXECUTE_INDEXED;');
      CALL EXECUTE
         ('SET SUGI27.COMPANY_INDEXED;');
      CALL EXECUTE
         ('WHERE VAT_NUMBER IN (');
   END;
   CALL EXECUTE (QUOTE (VAT_NUMBER));
   IF EOF THEN DO;
      CALL EXECUTE (');');
      CALL EXECUTE ('RUN;');
   END;
RUN;
```

SQL INTO CLAUSE

Last but not least, the seventh technique, which is also less known but proven to be very efficient, is the INTO clause with the SEPARATED BY keyword in the SELECT statement of the SQL procedure. This form of the INTO clause takes the values of a column and concatenates them into one macro variable. Always keep in mind that the maximum length of a macro variable value is 32K characters. This macro variable is then referenced in the WHERE statement to subset the lookup SAS data set.

This technique is especially appropriate when the lookup file is a large DBMS table and you use the SQL Procedure Pass-Through Facility to retrieve data from the DBMS table. The PROC SQL Pass-Through facility passes an SQL query from a SAS program directly to the DBMS, which performs the processing and returns the query result to SAS software for display or further analysis.

Use the SQL procedure to create the macro variable named VAT_LIST that contains a list of all VAT numbers that exist in the primary SAS data set SUGI27.BAD_DEBTOR. Each VAT number should be enclosed in double quotes and separated from the next by a comma.

PROGRAM 7-A

```
PROC SQL NOPRINT;
   SELECT DISTINCT VAT_NUMBER
          INTO :VAT_LIST SEPARATED BY '", "'
      FROM SUGI27.BAD_DEBTOR;
QUIT;
```

Because each VAT number only appears once in the SAS data set SUGI27.BAD_DEBTOR, you could omit the DISTINCT keyword.

By default, the results of a SELECT statement are displayed in the OUTPUT window. The NOPRINT option in the PROC SQL statement is useful when you are selecting values from a table into a macro variable and you do not want anything to be displayed in the OUTPUT window.

```
%LET VAT_LIST = &VAT_LIST;
```

This form of the INTO clause does not trim leading or trailing blanks. The %LET statement removes any leading or trailing blanks that may be stored in the macro variable value.

Use the new macro variable VAT_LIST to subset the lookup SAS data set SUGI27.COMPANY.

```
DATA SQL_INTO;
   SET SUGI27.COMPANY;
   WHERE VAT_NUMBER IN ("&VAT_LIST");
RUN;
```

The SYMBOLGEN system option writes the results of resolving the macro variable reference &VAT_LIST to the LOG window. Note that the opening quote is missing for the first VAT number and the closing quote is missing for the last VAT number. Enclose the macro variable reference &VAT_LIST in double quotes in the WHERE Statement to add these missing quotes. Make sure that you use double quotes. The word scanner continues to tokenize literals enclosed in double quotes, permitting the macro variable to resolve. The word scanner does not tokenize literals enclosed in single quotes, so the macro variable would not resolve.

Repeat the previous DATA step but use the indexed SAS data set SUGI27.COMPANY_INDEXED instead of the SAS data set SUGI27.COMPANY. Verify in the LOG window whether the index VAT_NUMBER was selected for WHERE clause optimization.

PROGRAM 7-B

```
PROC SQL NOPRINT;
   SELECT DISTINCT VAT_NUMBER
          INTO :VAT_LIST SEPARATED BY '", "'
      FROM SUGI27.BAD_DEBTOR;
QUIT;

%LET VAT_LIST = &VAT_LIST;

DATA SQL_INTO_INDEXED;
   SET SUGI27.COMPANY_INDEXED;
   WHERE VAT_NUMBER IN ("&VAT_LIST");
RUN;
```

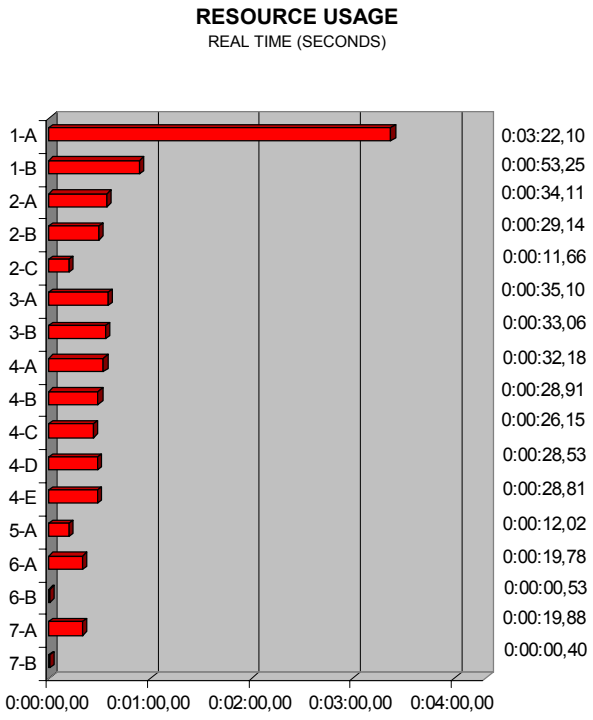
CONCLUSION

To determine which technique is the most efficient, we measured and compared the resource usage of each technique. We ran each program 5 times on a Windows NT PC and based our conclusions on averages, not on one individual execution.

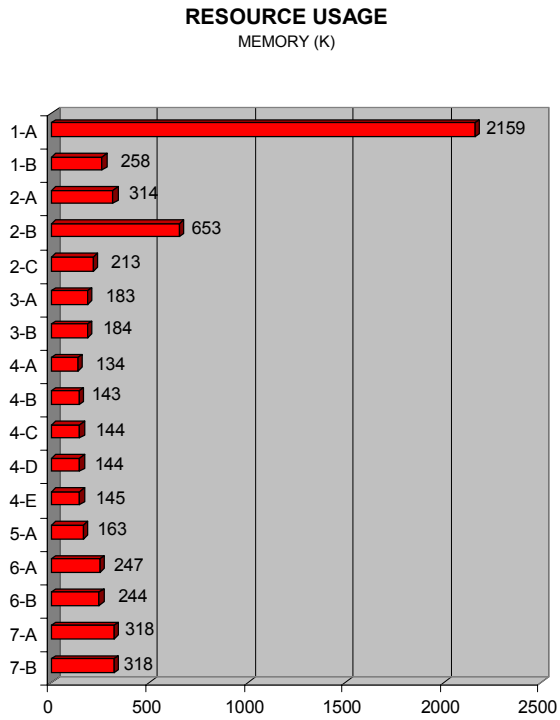
The following table summarizes the average real time needed by each program ranked from low to high:

PROGRAM NUMBER	INDEX ON LOOKUP SAS DATA SET?	REAL TIME (SECONDS)
7-B	YES	0:00:00.40
6-B	YES	0:00:00.53
2-C	YES	0:00:11.66
5-A	YES	0:00:12.02
6-A	NO	0:00:19.78
7-A	NO	0:00:19.88
4-C	NO	0:00:26.15
4-D	YES	0:00:28.53
4-E	YES	0:00:28.81
4-B	NO	0:00:28.91
2-B	NO	0:00:29.14
4-A	NO	0:00:32.18
3-B	YES	0:00:33.06
2-A	NO	0:00:34.11
3-A	NO	0:00:35.10
1-B	YES	0:00:53.25
1-A	NO	0:03:22.10

The following graph summarizes the average real time needed by each program in the order they were discussed in this paper:



The following graph summarizes the average memory needed by each program in the order they were discussed in this paper:



The following table summarizes the average memory needed by each program ranked from low to high:

PROGRAM NUMBER	INDEX ON LOOKUP SAS DATA SET?	MEMORY (K)
4-A	NO	134
4-B	NO	143
4-C	NO	144
4-D	YES	144
4-E	YES	145
5-A	YES	163
3-A	NO	183
3-B	YES	184
2-C	YES	213
6-B	YES	244
6-A	NO	247
1-B	YES	258
2-A	NO	314
7-A	NO	318
7-B	YES	318
2-B	NO	653
1-A	NO	2159

We can conclude that it is often more efficient to subset the large lookup SAS data set based on the list of unique key values that exist in the primary SAS data set instead of combining the two SAS data sets. The INTO clause with the SEPARATED BY keyword in the SELECT statement of the SQL procedure and the CALL EXECUTE routine have proven to be the most efficient. Keep in mind that both techniques are only appropriate if the list of unique key values that exist in the primary SAS data set is limited in size.

We can also conclude that in general the ease of writing programs is often inversely proportional to its efficiency. In other words, fairly straightforward programs often consume more computer resources than more complicated programs. However, writing programs that use fewer computer resources, but use more complex programming techniques, may result in an increase in development time. You should consider spending more development time to reduce computer resource usage in programs that are run frequently or process large amounts of data.

REFERENCES

SAS Institute Inc., *Combining and Modifying SAS Data Sets: Examples, Version 6, First Edition*, Cary, NC: SAS Institute Inc., 1995. 197 pp.

TRADEMARKS

CONTACT INFORMATION

Your comments and questions are valued and encouraged. Please feel free to contact the authors at:

Nancy CROONEN
CC Training Services BVBA
Kesseldallaan 12/202
B-3010 KESSEL-LO
BELGIUM
Work Phone: +32 496 28 45 28
Fax: +32 2 706 03 09
Email: nancy.croonen@solidpartners.be
Web: www.solidpartners.be

Henri THEUWISSEN
SOLID Partners NV
Minervastraat 14 bis
B-1930 ZAVENTEM
BELGIUM
Work Phone: +32 495 54 52 53
Fax: +32 2 706 03 09
Email: henri.theuwissen@solidpartners.be
Web: www.solidpartners.be