PhUSE 2007

Paper CS06

Comparison of different ways using table lookups on huge tables

Ralf Minkenberg, Boehringer Ingelheim Pharma GmbH & Co. KG, Ingelheim, Germany

ABSTRACT

In many application areas the problem to extract specific data records out of a huge master table based on key values and stored in a (smaller) lookup table is very common. A master table could for example contain common customer information in a bank data warehouse, or could contain basic demographic data about patients in a clinical study or project. Especially for mega trials or meta analysis these master tables could become really large.
The classical solutions by way of sorting or indexing the data records are compared with other more efficient methods, which lead to the benefit of savings in time and memory. The methods presented alternatively on the one hand are using well-known SAS® statements, on the other hand the new hash sort in SAS® version 9 is explained as well which provides new powerful possibilities for a quick and efficient lookup.

INTRODUCTION

In many problems which are solved in SAS® programming often a huge master table is given containing all interesting observations. For a concrete task then often only a very small part of these observations should be selected depending on information provided by another lookup table. For example in large clinical studies (mega trials) often some subgroup analyses must be performed selecting only specific patients out of the whole population.
The methods solving this lookup problem are, especially for huge datasets, very time and memory consuming and take most of the time a SAS® program needs at all. Therefore, optimizing the underlying SAS® code becomes an important issue for successful programming. This is even more important in a project where many similar problems occur and the program seems to run nearly endless. The time many SAS® programs need to finish can be shortened significantly if you use SAS® statements in an efficient manner. But the possibilities given within SAS® to do a lookup are often not known in general or in detail and – instead of trying some rather uncommon code – time is wasted by using a more traditional set of SAS® statements.
In the following sections seven methods to perform a table lookup within SAS® are explained and compared using as an example a lookup table with about 40,000 observations and a master table containing about 5 million observations with general information. In both tables, named LARGE and SMALL, respectively, the primary key (only one key variable is needed) values are unique. The size of the tables should ensure that differences in time and memory consumption are clearly seen. But even for smaller tables the knowledge of alternative lookup methods could be helpful especially if you have to perform very many lookups in one program.
The following seven methods are presented and explained:

1. DATA step with MERGE
2. Creating an index
3. PROC SQL
4. The FORMAT procedure and the PUT statement
5. SET statements with the KEY option
6. CALL EXECUTE
7. Hash sort

All introduced methods only need SAS® BASE statements, hash sort is implemented in SAS® version 9.
The different methods will be compared by looking at the consumed system time, the user CPU time, the real time, and the used memory (all information are available in the LOG after using the option FULLSTIMER).

TRADITIONAL METHODS

DATA STEP WITH MERGE

The most known and nearly everyday used method is a merge of the involved tables using a DATA step after sorting both tables by the key variable (here named NO). A corresponding SAS® program could look like the following:

```
PROC SORT DATA=large; BY no; RUN;

PROC SORT DATA=small; BY no; RUN;
```

```
DATA look;
  MERGE small(IN=a) large;
  BY no;
  IF a;
RUN;
```

Using the IN variable the needed observations from our lookup table SMALL are identified in the master table LARGE and are stored in the result table LOOK. All observations of the two datasets are read sequentially and then in a next step are filtered in the defined way.
Despite this very standard method being the preferred code for many everyday problems (with more smaller datasets used) it is the most time and memory consuming way to perform a table lookup.

DATA STEP WITH MERGE AND AN INDEX

To avoid the sorting of the huge master table LARGE in the above example, which is even more important if this must be done several times, you could create a corresponding index. This could be done for example with the procedure DATASETS:

```
PROC DATASETS LIBRARY=work MEMTYPE=data NOLIST;
  MODIFY large;
    INDEX CREATE no / UNIQUE;
  RUN;
QUIT;
```

The UNIQUE option controls that every value of the key variable NO is allowed to appear only once.
To perform a merge between our two tables now only the smaller lookup table has to be sorted:

```
PROC SORT DATA=small; BY no; RUN;

DATA look;
  MERGE small(IN=a) large;
  BY no;
  IF a;
RUN;
```

The creation of an index for huge tables consumes much time. Therefore, this method should only be used if many operations are planned where the index variable is involved. For one lookup as in our example we will get no reduction of running time, but the memory needed is smaller.

PROC SQL

Alternatively to a DATA step a table lookup can also be executed with the procedure SQL. The very basic code could look like the following:

```
PROC SQL;
  CREATE TABLE look AS
         SELECT large.* FROM small, large
         WHERE small.no = large.no;
QUIT;
```

Here no sorting of the two tables is required, but for optimization a sort of both tables as well as creating an index on the master table LARGE could be done. The above SQL statements perform a so called “inner join” between two tables, which results in the Cartesian product of the tables restricted to the observations satisfying the WHERE clause. If no sorting or index creation is done, this method could take a long time.
If you want to use SQL, a table lookup can be performed in a more efficient (quicker) way if you replace the “inner join” with a corresponding subquery:

```
PROC SQL;
  CREATE TABLE look AS
         SELECT * FROM large
         WHERE no IN (SELECT no FROM small);
QUIT;
```

This subquery which is a part of another outer query avoids the calculation of a Cartesian product. Only rows of a table are selected with identical to values of another table. Sometimes an appropriate index will reduce time even more.

OTHER EFFICIENT METHODS

In addition to the methods introduced in the previous section there are many other possibilities to perform a table lookup in SAS®, which are not so well-known but could save time and memory (see also [1]). As such a first possibility a method using PROC FORMAT.

THE FORMAT PROCEDURE AND A PUT STATEMENT

First a dataset for formatting is created dynamically out of our lookup table, which in a second step is used to read the needed observations out of our master table. The corresponding SAS® code looks like the following:

```
DATA fmtin;
  SET small (RENAME=(no=start));
  RETAIN label 'SMALL' fmtname 'NO';
RUN;

PROC FORMAT CNTLIN=fmtin;
RUN;

DATA look;
  SET large;
  WHERE PUT(no, no.) EQ 'KLEIN';
RUN;
```

The WHERE statement only selects the actually needed observations of the master table LARGE and, therefore, these huge master table is not read in its full size.

SET STATEMENT WITH THE KEY OPTION

The following method for table lookup is only usable if an index is created, but then it is very efficient, even more for huge tables. In a first SET statement the smaller lookup table will be read sequentially. Afterwards in a second SET statement the indexed master table is read, selecting only the wanted observations using the KEY option. This option ensures that only matching values for the key variable in the two tables will be taken:

```
DATA look;
  SET small;
  SET large KEY=no;
RUN;
```

If the lookup table SMALL contains many identical values for the key variable, the results you get with this method are not always what you would expect. So only with unique key values also in the smaller lookup table this method is recommended.

CALL EXECUTE

Another efficient method could be realized with a CALL EXECUTE statement. Using a DATA _NULL_ step the necessary SAS® code will be produced in such a way that all unique values of the key variable of the lookup table will be listed in an IN operator within a DATA step:

```
DATA _null_;
  SET small END=last,
  IF _n_ EQ 1 THEN DO;
    CALL EXECUTE('DATA look;');
    CALL EXECUTE('SET large;');
    CALL EXECUTE('WHERE no IN (');
  END;
  CALL EXECUTE(no);
  IF last THEN DO;
    CALL EXECUTE(');');
    CALL EXECUTE('RUN;');
  END;
RUN;
```

The SAS® code produced by CALL EXECUTE will be executed automatically as “normal” SAS® code after the DATA _NULL_ step. Although this method could be efficient for a table lookup you must be aware that SAS® will produce

an error message and terminate the DATA step if too many values of your key variable are present in the lookup table (`ERROR: An internal expression limit of 32767 entries in a list has been reached. This expression cannot be parsed.`). It is nevertheless possible to split the list of values over many data step lines using for example IF statements to avoid this problem.

HASH SORT

THE BASICS OF HASHING

Generally hashing defines a procedure which maps a relatively long key value (of arbitrary form) to a smaller number out of a relatively small set of integers using special mathematical algorithms.
The basic principle is demonstrated by an easy example. In a table there are 10 observations identified by a three digit numeric key value. For example:

```
DATA example;
  INPUT key @@;
  CARDS;
185 971 400 260 922 970 543 532 050 067
  ;
RUN;
```

In an ideal situation it would be nice to find a hash function which maps the given three digit key values one-to-one to the numbers 0 to 9, this would yield a perfect hash function. The derivation of such a perfect hash function is in general very lavishly, difficult to code and, if only one additional observation appears, has to be derived completely new. Therefore, in practice so called non-perfect hash functions are used. Such a function could be calculated easier, but no one-to-one relationship between the original key value and the hash value can be guaranteed. An algorithm how to handle such possible “collisions” must be given, too. An often used easy example for a hash function is the division of the given key values by a prime number and taking the rest of the division (modulo).
There are two questions to be answered if you want to use hashing in an effective way:
1. Which part of the possible range of integers should be actually filled with values after hashing? The less of the possible numbers are given after hashing the less is the probability of collisions, but the more memory is necessary. For practical reasons a filling of about 80% seems to be reasonable.
2. If the proposed portion of values after hashing is set the optimal range has to be calculated for the results of the hash function. For this problem known algorithms can be found.
If we want a fill portion of about 80% in our small example with some defined algorithms we get a hash size of 13. Therefore, we divide the key values by 13 and take the rest as the result. We get the following mapping:

KEY	HASH
185	04
971	10
400	11
260	01
922	13
970	09
543	11
532	13
050	12
067	03

We see two collisions in this example. Both the key values 400 as well as 543 are mapped to 11, and both 922 as well as 532 are mapped to 13. The nullification of this collision is performed by different well-defined algorithms. A method must be found, which find a new “free” integer for those key values which are mapped a second time to a hash number. (see [2])
Using hashing table lookups and searching is the most efficient method because every interesting observation can be found directly via its position in a table. In the next section we will demonstrate hashing in SAS®.

HASHING IN SAS®

To perform hashing for a table lookup in SAS version 9 object-oriented code in a DATA step is needed. For the two tables introduced in section “Traditional methods” the code looks like the following:

```
DATA look(DROP=rc);
  LENGTH no $12 large $8;
  DECLARE Hash ha();
  rc = ha.DefineKey('no');
  rc = ha.DefineDone();
```

```
  DO UNTIL (last1);
    SET small END=last1;
    rc = ha.add();
  END;
  DO UNTIL (last2);
    SET large END=last2;
    rc = ha.find();
    IF rc EQ 0 THEN OUTPUT;
  END;
  STOP;
RUN;
```

After setting the length of the necessary data elements with the LENGTH statement, the hash table HA will be defined using DECLARE. With the DefineKey method the key variable is given and DefineDone finishes the initialization of the hash object. Using the Add method a key value from SMALL is loaded into the hash table. Now with the Find method matching key values in LARGE are searched and saved.
The perhaps a little bit longish code could be rewritten somewhat shorter:

```
DATA look;
  SET small POINT=_n_;
  DECLARE Hash ha(dataset:'work.small', hashexp:10);
  ha.DefineKey('no');
  ha.DefineDone();
  DO UNTIL (last);
    SET large END=last;
    IF ha.find() EQ 0 THEN OUTPUT;
  END;
  STOP;
RUN;
```

All needed definitions are set by reading one row from SMALL. The DECLARE statement is broadened by stating the lookup dataset as well as the maximal number of hash objects (here 2**10=1024).
Starting with version 9 in SAS® the new implemented hash function enables an efficient table lookup and reduces the needed time and memory resources significantly. There are more possibilities using the hash function in SAS®, but more details and detailed code would blow up the contents of this paper. More can be found in the literature. (see [3]).

COMPARISION OF THE DIFFERENT METHODS

The time and memory consumption of the methods introduced above are shown in the following. For all example programs the dataset SMALL with about 40,000 observations and one numeric key variable and the dataset LARGE with about 5,000,000 observations, the key variable and three additional variables are used. From the LOG (given the option FULLSTIMER) the system CPU time, the user CPU time, the real time, and memory have been taken for evaluation. The different methods are enumerated in the following way:

1 DATA step with MERGE
2 DATA step with MERGE and an index
4 PROC SQL inner join
6 PROC SQL subquery
8 FORMAT procedure and PUT statement
9 SET statement with KEY option
11 Hashing

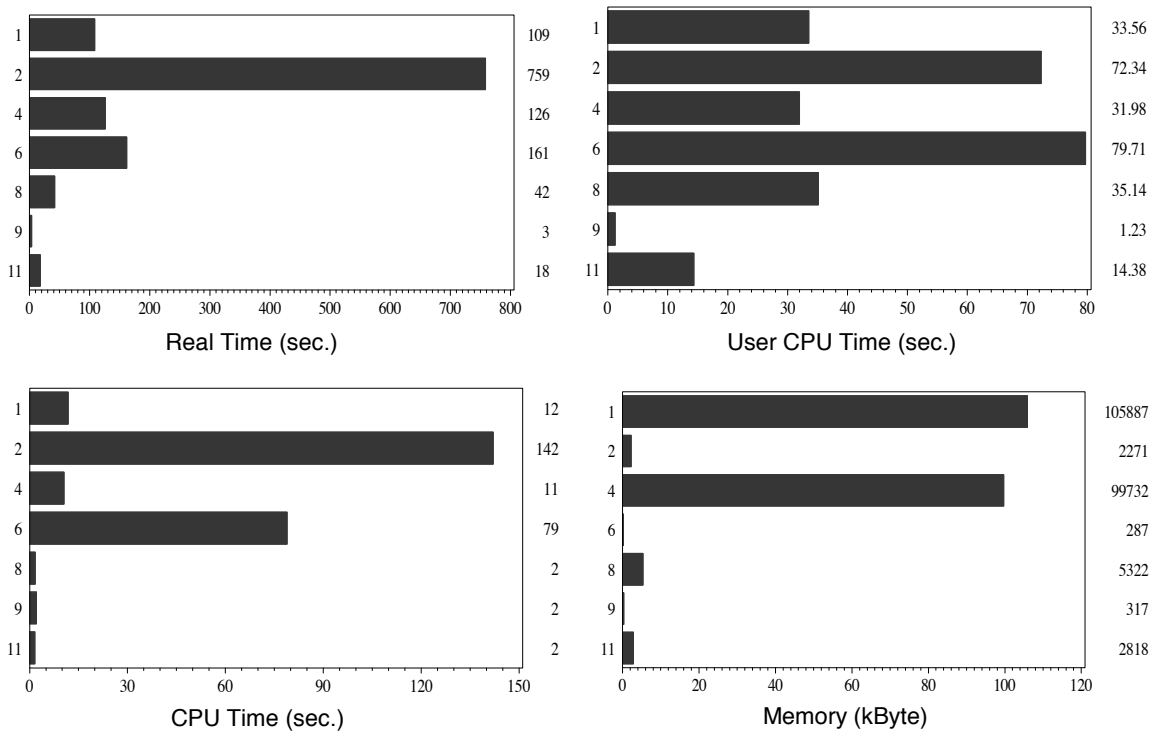

Figure 1: Time and memory consumption of different lookup methods

The hash method (not needing an index) and the method using SET and the KEY option (needing an index) are the most efficient methods. Especially, both are significantly better than the traditional methods used in most situations.

CONCLUSION

Table lookups are a common problem in nearly any field of SAS® programming and take most of the time and memory a program needs. Besides the traditional well-known methods there are many alternative coding solutions possible to solve a table lookup significantly faster and with less memory. This can decrease the time your SAS® programs needs to finish and therefore, save time and money.

REFERENCES

[1] Croonen, N., Theuwissen, H. (2002): *Table Lookup: Techniques Beyond the Obvious*, Paper 11-27, SUGI 27
[2] Dorfman, P., Snell, G. (2002): *Hashing Rehashed*, Paper 12-27, SUGI 27
[3] Dorfman, P., Vyverman, K. (2004): *Hash Components Objects: Dynamic Data Storage and Table Look-Up*; Paper 238-29, SUGI 29

CONTACT INFORMATION

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

Ralf Minkenberg
Boehringer Ingelheim Pharma GmbH & Co. KG
Binger Str. 173
55216 Ingelheim / Germany
Work Phone: +49-(0)6132-7797209
Fax: +49-(0)6132-7297209
Email: ralf.minkenberg@boehringer-ingelheim.com
Web: http://www.boehringer.com