Paper 242-2011

Don’t Waste Too Many Resources to Get Your Data in a Specific Sequence

Henri THEUWISSEN, BI Knowledge Sharing, Belgium

ABSTRACT

Many applications contain too many PROC SORT steps, and sorting data is CPU and space consuming.

This presentation compares different sorting methods, the SORT procedure, the SQL procedure, and sorting in HASH tables, and compares the results in performance.

Tips are given on how to remove redundant sorts by using the SORTEDBY option, the NOTSORTED option, or the implicit sort of procedures such as PROC SUMMARY and PROC REPORT.

Also, new options in SAS® 9.2 enable you to get the data in the required sequence without extra DATA steps.

All techniques are presented with examples and resource usage figures.

INTRODUCTION

Data and information is almost always presented in a sorted or a structured way. The presentation of information must be intuitive, easy to understand or easy to analyze. This requires that tables must be sorted several times to prepare the data or to create the information.

Developers often write – obsolete – PROC SORT steps to make sure that the data is in the correct sequence for a subsequent DATA or PROC step. Developers often justify these PROC steps with the – incorrect – assumption “If the data is sorted, SAS will not sort again”. This is only partly correct.

The paper provides an answer to two questions:

- How to get the best performance of sorting?
- How to avoid SORT steps?

In the performance improvement possibilities, the paper covers:

- A comparison of different methods to sort data: the SORT and SQL procedures and the HASH object in DATA steps.
- The best method to create sorted output with unique combinations of the BY variables.
- Options that reduce the resources (CPU time, real time, memory) in sorting.

Tips to remove redundant SORT steps or to avoid additional DATA steps are presented by:

- Exploiting the implicit sort of several procedures.
- Options to avoid the execution of a PROC SORT step.
- Options to get an intuitive sequence as result of a SORT procedure.

COMPARING SORT METHODS

SORT Procedure – SQL Procedure – HASH Object

SAS provides many methods to get your data in a specific sequence. The most familiar ways are the SORT and the SQL procedure. The HASH object in DATA step processing also offers a solution to get your data sorted.

Within the next example these three methods are compared by means of resource requirements. The three methods are used to sort a SAS table on two variables. The same methods are executed on different SAS tables with the same structure, but with the number of observations varying from 1 million to 100 million.

Notice the MULTIDATA option in the HASH object definition to allow duplicate key combinations.

```
*--------------------*;
*   SORT Procedure   *;
*--------------------*;

proc sort data = sgf.invoices
          out = invoices_sort;
   by doc_nr
      line_nr;
run;


*-------------------*;
*   SQL Procedure   *;
*-------------------*;

proc sql;
   create table invoices_sql as
      select *
         from sgf.invoices
         order by doc_nr,
                  line_nr;
quit;


*-----------------*;
*   HASH Object   *;
*-----------------*;

data _null_;
   if _n_ = 0 then set sgf.invoices;
   if _n_ = 1 then do;
      declare hash ht (dataset: "sgf.invoices",
                       ordered: "A",
                       multidata: "Y");
      ht.definekey    ("doc_nr",
                       "line_nr");
      ht.definedata   ("client",
                       "doc_nr",
                       "line_nr",
                       "posting_date",
                       "consumption",
                       "clearing_date",
                       "doc_type",
                       "clearing_reason");
      ht.definedone   ( );
   end;
   ht.output (dataset: "invoices_hash");
run;
```

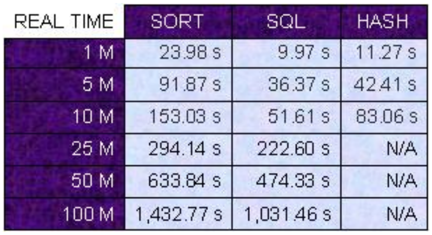
The resource statistics show that:

- The HASH object can treat only a limited table size, due to memory limitations. In our example, the following error occurred:

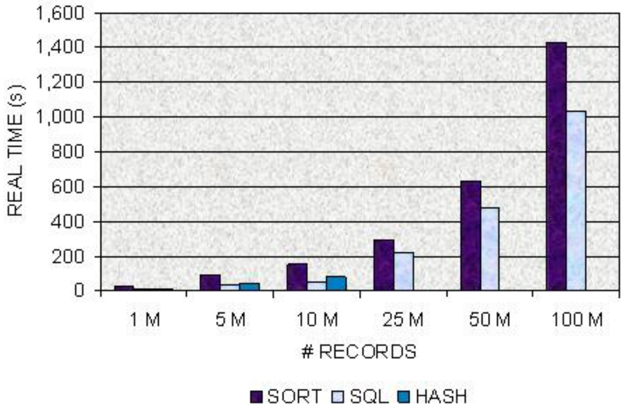
```
ERROR: Hash object added 12058608 items when memory failure occurred.
```

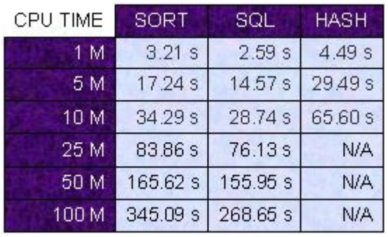
- The SQL procedure performs better than the SORT procedure, both on real time and on total CPU time. Notice that in SAS® 9.2 the performance of the SQL procedure is improved a lot compared to previous releases.

- Memory requirements for the SORT and the SQL procedure are independent of the number of observations; the memory requirements in the HASH object depend on the table size.
- The HASH object requires more CPU time than the SORT or the SQL procedure.

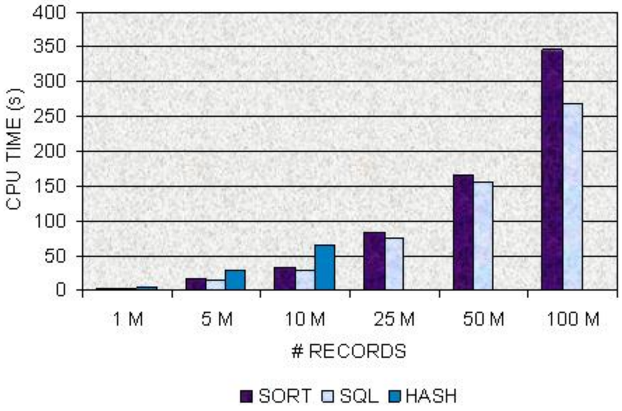


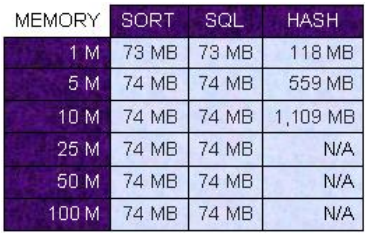
REAL TIME	SORT	SQL	HASH
1 M	23.98 s	9.97 s	11.27 s
5 M	91.87 s	36.37 s	42.41 s
10 M	153.03 s	51.61 s	83.06 s
25 M	294.14 s	222.60 s	N/A
50 M	633.84 s	474.33 s	N/A
100 M	1,432.77 s	1,031.46 s	N/A



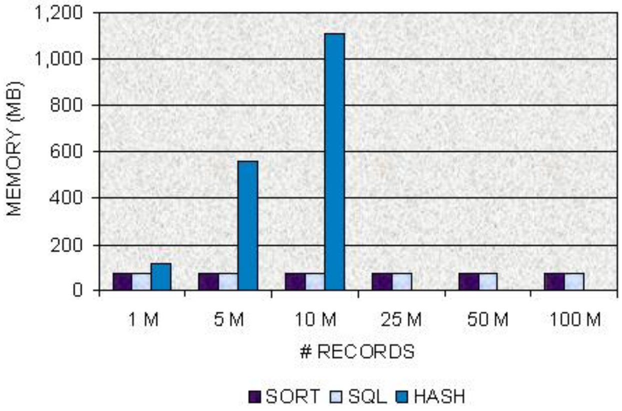


CPU TIME	SORT	SQL	HASH
1 M	3.21 s	2.59 s	4.49 s
5 M	17.24 s	14.57 s	29.49 s
10 M	34.29 s	28.74 s	65.60 s
25 M	83.86 s	76.13 s	N/A
50 M	165.62 s	155.95 s	N/A
100 M	345.09 s	268.65 s	N/A





MEMORY	SORT	SQL	HASH
1 M	73 MB	73 MB	118 MB
5 M	74 MB	74 MB	559 MB
10 M	74 MB	74 MB	1,109 MB
25 M	74 MB	74 MB	N/A
50 M	74 MB	74 MB	N/A
100 M	74 MB	74 MB	N/A



ELIMINATING DUPLICATE BY COMBINATIONS

SORT Procedure – HASH Object

The NODUPKEY option in the PROC SORT statement checks for and eliminates observations with duplicate BY values.

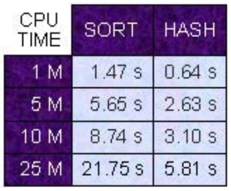
Initially HASH objects in DATA step processing required unique key combinations. From SAS® 9.2 onwards, duplicate key combinations are supported. By default, the HASH object will store only unique key combinations. A DATA step with a HASH object can be used to build a similar result as with the NODUPKEY option in the SORT procedure.

The following example compares both methods on a SAS table with 8 variables, sorting on 1 variable. The SORT procedure and the DATA step are executed on SAS tables with the same structure but with a different number of observations.

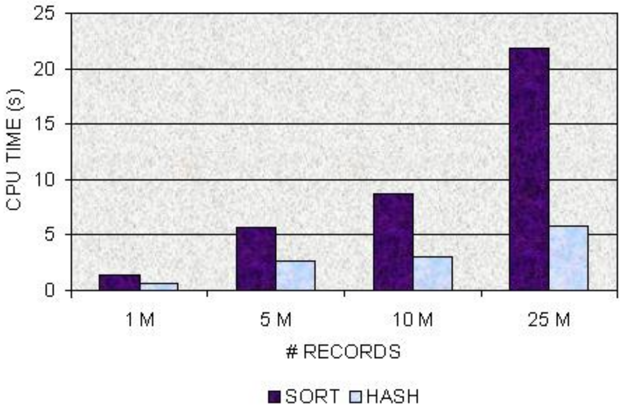
```
*---------------------*;
*    SORT Procedure   *;
*---------------------*;

proc sort data = sgf.invoices
          out = invoices_sort
          nodupkey;
   by client;
run;


*------------------*;
*    HASH Object   *;
*------------------*;

data _null_;
   if _n_ = 0 then set sgf.invoices;
   if _n_ = 1 then do;
      declare hash ht (dataset: "sgf.invoices",
                       ordered: "A");
      ht.definekey     ("client");
      ht.definedata    ("client",
                       "doc_nr",
                       "line_nr",
                       "posting_date",
                       "consumption",
                       "clearing_date",
                       "doc_type",
                       "clearing_reason");
      ht.definedone    ( );
   end;
   ht.output (dataset: "invoices_hash");
run;
```

The code for the HASH object is more complex than the traditional PROC SORT statements, and the HASH object is limited by the memory, but the resource statistics show that:

- The HASH object is much faster than the SORT procedure.
- The HASH object requires fewer memory (550) than the SORT procedure (66930)



CPU TIME	SORT	HASH
1 M	1.47 s	0.64 s
5 M	5.65 s	2.63 s
10 M	8.74 s	3.10 s
25 M	21.75 s	5.81 s

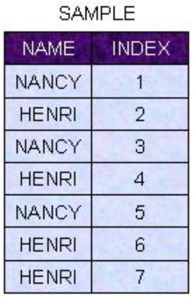


SORT RESOURCE OPTIMIZATION

The NOEQUALS Option

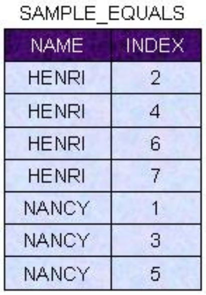
With the default execution of the SORT procedure, observations within BY groups keep the same relative order as within the original data set. You can specify an option within the PROC statement to explicitly request this behavior (EQUALS) or you can indicate that the sequence of the observations within a BY group can be in any order (NOEQUALS). The option NOEQUALS requires fewer resources than the – default – option EQUALS.

Consider a SAS table SAMPLE with 2 variables: NAME and INDEX. Sorting this table on the variable NAME results in a different output, depending on whether you specify the EQUALS or NOEQUALS option. Both results have a correct sequence for the variable NAME. The EQUALS option results in an output where the observations in each BY-group kept their relative position (Henri 2 – 4 – 6 – 7, Nancy 1 – 3 – 5) whereas the NOEQUALS option lost the original relative position (Henri 6 – 2 – 7 – 4, Nancy 3 – 1 – 5). It is obvious that keeping the original relative position will request more search operations.



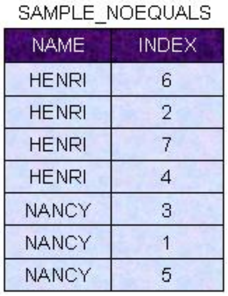
SAMPLE

NAME	INDEX
NANCY	1
HENRI	2
NANCY	3
HENRI	4
NANCY	5
HENRI	6
HENRI	7



SAMPLE_EQUALS

NAME	INDEX
HENRI	2
HENRI	4
HENRI	6
HENRI	7
NANCY	1
NANCY	3
NANCY	5



SAMPLE_NOEQUALS

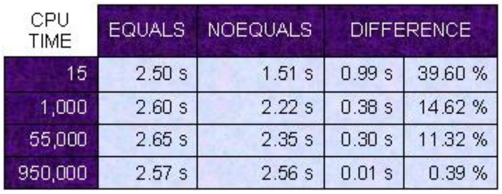
NAME	INDEX
HENRI	6
HENRI	2
HENRI	7
HENRI	4
NANCY	3
NANCY	1
NANCY	5

The impact of the EQUALS / NOEQUALS option is examined in the execution of the SORT procedure on a table with 1 million observations. To get an idea about the impact of the content of the table on the resource differences, the table is sorted on four different variables. The main difference between these variables is the number of unique values in the table (15 – 1,000 – 55,000 – 950,000).

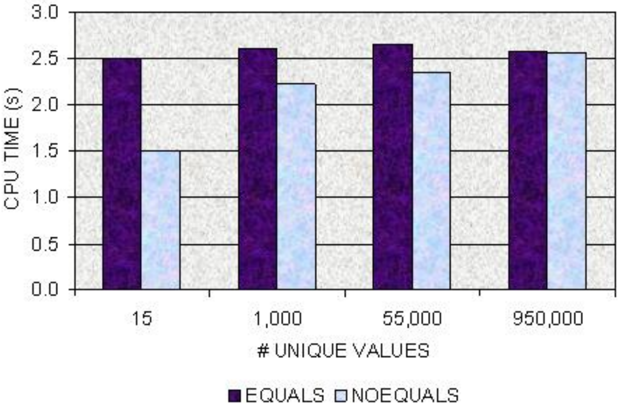
```
*---------------------------------------*;
*   SORT Procedure with EQUALS Option   *;
*---------------------------------------*;

proc sort data = sgf.invoices
          out = invoices_equals
          equals;
   by &variable;
run;


*-----------------------------------------*;
*   SORT Procedure with NOEQUALS Option   *;
*-----------------------------------------*;

proc sort data = sgf.invoices
          out = invoices_noequals
          noequals;
   by &variable;
run;
```

Note: the test was executed with four different values for the macro variable in the BY statement; each variable used in the BY statement has a different number of unique values in the table.

The resource statistics show that the NOEQUALS option is always faster than the – default - EQUALS option. The difference increases with a lower number of unique values.



CPU TIME	EQUALS	NOEQUALS	DIFFERENCE	
15	2.50 s	1.51 s	0.99 s	39.60 %
1,000	2.60 s	2.22 s	0.38 s	14.62 %
55,000	2.65 s	2.35 s	0.30 s	11.32 %
950,000	2.57 s	2.56 s	0.01 s	0.39 %



The TAGSORT Option

When sorting a large table, disk space or memory might become a problem. Using the TAGSORT option in the PROC SORT statement will execute the sort in two steps: First a sort is executed, using only the variables in the BY statement and the observation number, and the result is stored in the temporary utility files. Then the observation number is used to add the remaining variables from the base table. The option is useful when the total length of the variables in the BY statement is small, compared to the total observation length.

The effect of the TAGSORT option is examined on sorting a table with approx. 850,000 observations on three variables. A first test is executed on this table with a total of 170 variables and an observation length of 752 bytes. Afterwards a second test is executed on the table with only 20 variables and an observation length of 88 bytes.

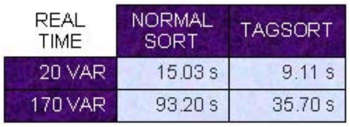
```
*-------------------------------------------*;
*   SORT Procedure without TAGSORT Option   *;
*-------------------------------------------*;

proc sort data = sgf.flight_details
          out = flight_details_sort;
   by confirmation_code
      first_name
      last_name;
run;


*----------------------------------------*;
*   SORT Procedure with TAGSORT Option   *;
*----------------------------------------*;

proc sort data = sgf.flight_details
          out = flight_details_tagsort
          tagsort;
   by confirmation_code
      first_name
      last_name;
run;
```

When using the TAGSORT option, a note is written to the Log, indicating that every observation is read twice.

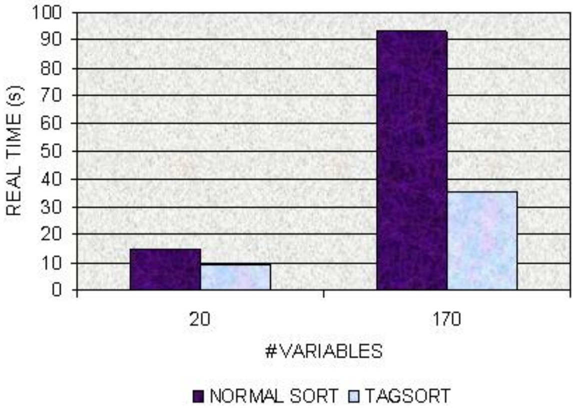
```
NOTE: Tagsort reads each observation of the input data set twice.
```

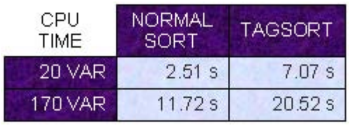
The resource statistics show that:

- The TAGSORT option results in an important decrease in memory requirements.
- The TAGSORT option consumes more CPU time.
- The TAGSORT option requires less real time.

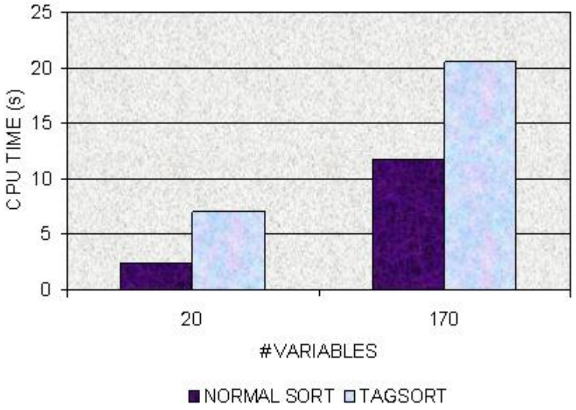


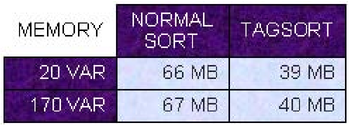
REAL TIME	NORMAL SORT	TAGSORT
20 VAR	15.03 s	9.11 s
170 VAR	93.20 s	35.70 s



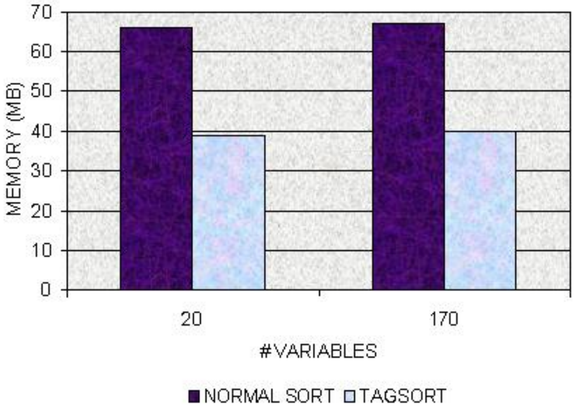


CPU TIME	NORMAL SORT	TAGSORT
20 VAR	2.51 s	7.07 s
170 VAR	11.72 s	20.52 s





MEMORY	NORMAL SORT	TAGSORT
20 VAR	66 MB	39 MB
170 VAR	67 MB	40 MB



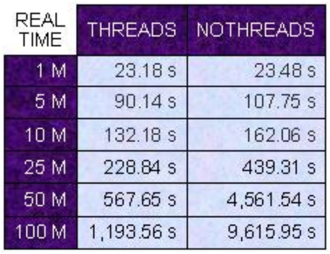
The THREADS Option

Parallel processing is supported in SAS since SAS® 9: it allows multiple CPUs to process simultaneously a task. This technology takes advantage of hardware that has multiple CPUs, called SMP (Symmetric MultiProcessor) machines. Performance improvements are achieved for both I/O and application processing.

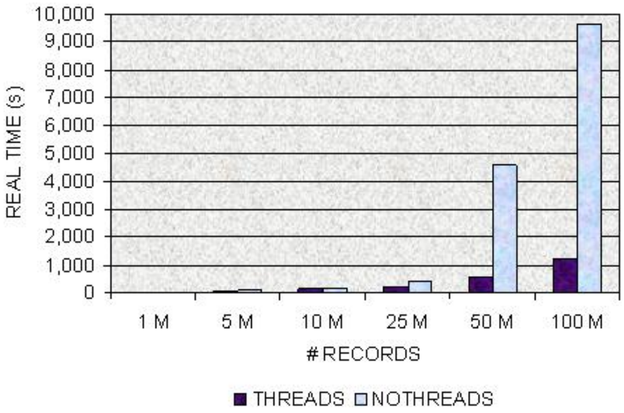
Parallel processing is requested by specifying the THREADS option. The THREADS or NOTHREADS option can be specified in a global OPTIONS statement or within a specific PROC statement. The default value is THREADS. Additionally you can specify the number of CPUs to use, using the CPUCOUNT option.

A PROC SORT is executed on a table with 12 variables. Both THREADS and NOTHREADS were examined. The procedure was executed on a PC with 2 processors.

The following table shows the real time for a SORT on a table with the same structure but with a different number of observations. Notice that THREADS shows an important performance improvement on large tables.



REAL TIME	THREADS	NOTHREADS
1 M	23.18 s	23.48 s
5 M	90.14 s	107.75 s
10 M	132.18 s	162.06 s
25 M	228.84 s	439.31 s
50 M	567.65 s	4,561.54 s
100 M	1,193.56 s	9,615.95 s

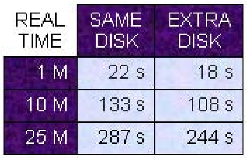


Separate Disks for Permanent and Temporary Data

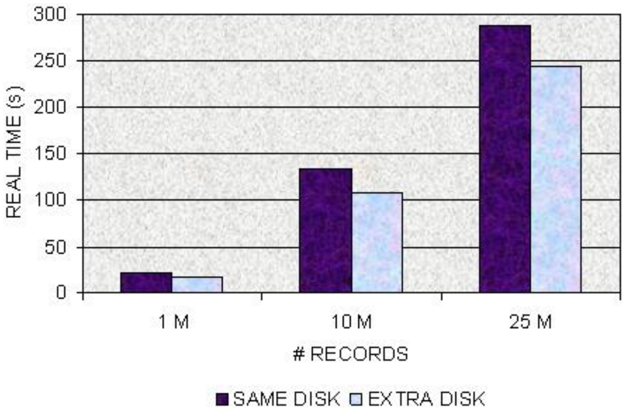
When examining the CPU usage of a process, you will find out that often the CPU is not fully used: the application waits for I/O operations to continue processing. Such a process is said to be I/O-bound.

A possible solution to improve the I/O performance is the usage of the SAS Scalable Performance Data (SPD) engine. The SPD engine partitions the data in partitioned data sets to improve I/O operations. These data sets can reside on one disk or on multiple disks.

Another solution is to split the permanent data and the WORK library on different volumes. The benefit of using multiple disks is illustrated in a SORT procedure execution on a table with the same structure but with a different number of observations. The sort is first executed with all data on the same disk, afterwards with the WORK library on a separate external disk.



REAL TIME	SAME DISK	EXTRA DISK
1 M	22 s	18 s
10 M	133 s	108 s
25 M	287 s	244 s



AVOIDING REDUNDANT SORTS

Examining any set of SAS programs will show a lot of PROC SORT steps. Many of these SORT steps are not required, and will just consume a lot of resources.

Obsolete Sorts Preceding other Procedures

Many procedures, like MEANS, SUMMARY, TABULATE and REPORT have a built-in functionality to return a report or data set in sorted order. They do not require a preceding sort of the data set.

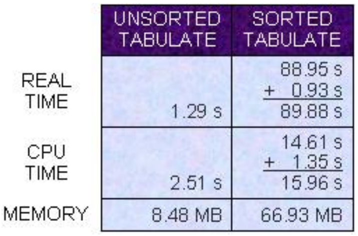
When a BY statement is used in the PROC step though, the data must be sorted on the variables in the BY statement or an index must exist on these variables.

Consider the following example, building a tabular report. The result is created on a table with 5 million observations. First the result is created by using the TABULATE procedure on the base table, which is not sorted, then the base table is sorted in a PROC SORT step, and the TABULATE is executed on the sorted result.

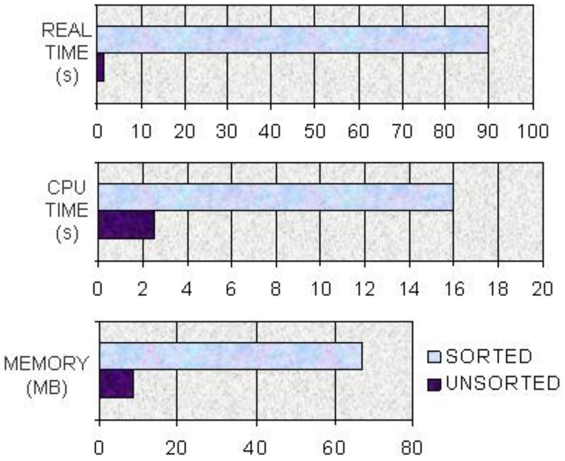
```
*-------------------------------------------*;
*   TABULATE Procedure with Unsorted Data   *;
*-------------------------------------------*;

proc tabulate data = sgf.invoices;
   class client posting_date;
   table client = ' ',
         posting_date = ' ' * n = ' ' * f = commax9.;
run;


*-----------------------------------------*;
*   TABULATE Procedure with Sorted Data   *;
*-----------------------------------------*;

proc sort data = sgf.invoices
          out = invoices_sorted;
   by client
      posting_date;
run;

proc tabulate data = invoices_sorted;
   class client posting_date;
   table client = ' ',
         posting_date = ' ' * n = ' ' * f = commax9.;
run;
```

The resource statistics show that:

- Using the procedure on sorted data is indeed faster than creating the report on unsorted data.
- The gain is lost completely by the preceding execution of the SORT step: the cost of the SORT procedure is more than 50 times the profit in the TABULATE procedure.
- Due to the SORT procedure, the second solution requires higher memory values.



	UNSORTED TABULATE	SORTED TABULATE
REAL TIME	1.29 s	88.95 s + 0.93 s 89.88 s
CPU TIME	2.51 s	14.61 s + 1.35 s 15.96 s
MEMORY	8.48 MB	66.93 MB



The Sort Flag

To avoid unnecessary sorting of your SAS tables, SAS stores a sort flag in the descriptor portion of the tables when a SORT or SQL procedure is executed. Additionally, information is kept about the variables used in the sort and about the collating sequence.

```
                         Sort Information

                    Sortedby        DOC_NR LINE_NR
                    Validated       YES
                    Character Set   ANSI
```

When you execute a PROC SORT step, SAS will check the sort flag, and will not execute a sort, when the data set is already sorted on the specified variable(s). A message is printed in the Log.

```
NOTE: Input data set is already sorted, no sorting done.
```

When you execute a PROC SORT to store the result in a new table, SAS will check the sort flag, and will not execute a sort, when the data set is already sorted on the specified variable(s). SAS will copy the input table in the output table and a message is printed in the Log.

```
NOTE: Input data set is already sorted; it has been copied to the output data set.
```

Notice that the SORT procedure and the SQL procedure set the sort flag, while sorting using the HASH object does not set the sort flag. Also, procedures like MEANS and SUMMARY create an output table that is sorted. Examining the descriptor portion of such an output table shows that no sort flag is set.

When you process a sorted table with a DATA step, the sort flag is lost. Notice that this is quite normal, since the values of the variables – on which the table is sorted – can change in the DATA step.

The following table summarizes what happens when you use a PROC SORT on a table that is created with different methods, and which contains sorted data.

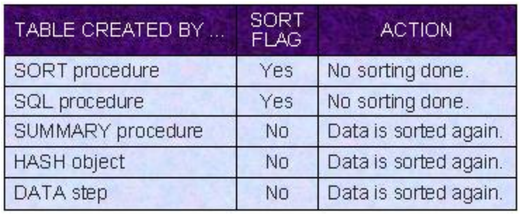


TABLE CREATED BY ...	SORT FLAG	ACTION
SORT procedure	Yes	No sorting done.
SQL procedure	Yes	No sorting done.
SUMMARY procedure	No	Data is sorted again.
HASH object	No	Data is sorted again.
DATA step	No	Data is sorted again.

The SORTEDBY Option

The SORTEDBY option can be specified when creating a SAS table, to set the sort flag to YES.

When specifying the SORTEDBY option SAS will:

- Set the sort flag to YES in the descriptor portion of the data set.
- Indicate that the sort is not validated by the system.
- Use the variables specified in the SORTEDBY option to create the descriptor information.

When specifying the SORTEDBY option the SAS system will NOT:

- Sort the data set when a PROC SORT is executed.
- Verify whether the data set is sorted on the variables specified in the SORTEDBY option.

Example

```
data invoices_hash_flag (sortedby = doc_nr line_nr);
   set invoices_hash;
run;
```

```
                                   Sort Information

                              Sortedby       DOC_NR LINE_NR
                              Validated      NO
                              Character Set  ANSI
```

The PRESORTED Option

The PRESORTED option was introduced with SAS® 9.2. The PRESORTED option in the PROC statement checks if the input data set is sorted already, even if the sort flag is not set. This option can avoid sorting a sorted data set, which does not have the sort flag.

A test is executed on a table with 1 million observations and 8 variables. The table is already sorted on 1 variable. The table is used in a PROC SORT step with and without the PRESORTED option. The same test is executed on a table with the same structure but with 5 million observations. Finally a SORT procedure with the PRESORTED option is executed on the 2 tables, where the table was not sorted yet.

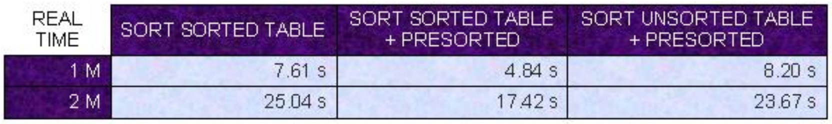
```
proc sort data = invoices_hash
          out = invoices_presorted
          presorted;
   by doc_nr;
run;
```

A note is written to the Log about the result of the check on the input data set.

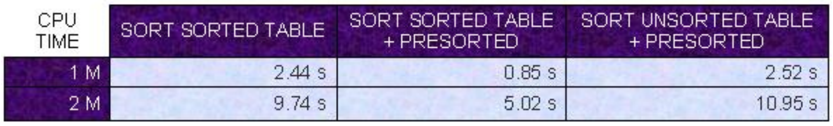
```
NOTE: Sort order of input data set has been verified.
NOTE: Input data set is already sorted; it has been copied to the output data set.
```

The resource statistics show that the PRESORTED option:

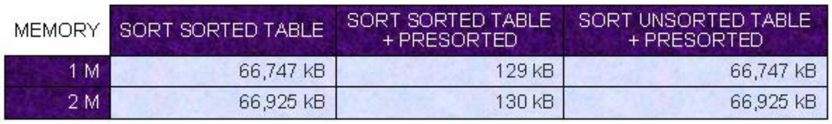
- Can result in an important reduction of real time, CPU time and memory requirements.
- Requires more resources if the table is not sorted yet.



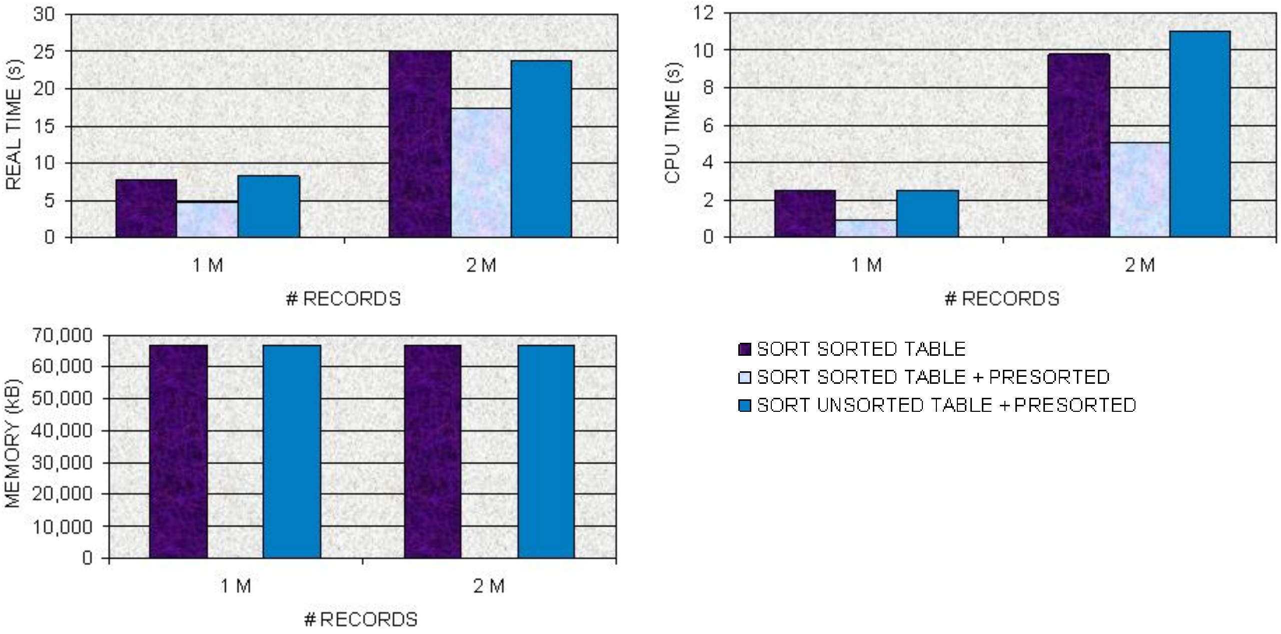
REAL TIME	SORT SORTED TABLE	SORT SORTED TABLE + PRESORTED	SORT UNSORTED TABLE + PRESORTED
1 M	7.61 s	4.84 s	8.20 s
2 M	25.04 s	17.42 s	23.67 s



CPU TIME	SORT SORTED TABLE	SORT SORTED TABLE + PRESORTED	SORT UNSORTED TABLE + PRESORTED
1 M	2.44 s	0.85 s	2.52 s
2 M	9.74 s	5.02 s	10.95 s

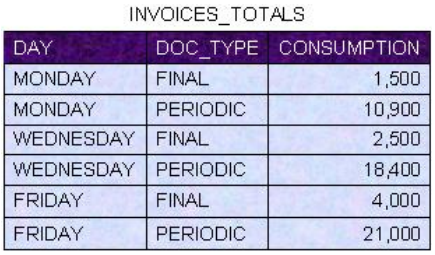


MEMORY	SORT SORTED TABLE	SORT SORTED TABLE + PRESORTED	SORT UNSORTED TABLE + PRESORTED
1 M	66,747 kB	129 kB	66,747 kB
2 M	66,925 kB	130 kB	66,925 kB



The NOTSORTED Option

Consider a table with data consolidated by weekday and another variable. The table has no sort flag.



INVOICES_TOTALS

DAY	DOC_TYPE	CONSUMPTION
MONDAY	FINAL	1,500
MONDAY	PERIODIC	10,900
WEDNESDAY	FINAL	2,500
WEDNESDAY	PERIODIC	18,400
FRIDAY	FINAL	4,000
FRIDAY	PERIODIC	21,000

Specifying a BY statement with the variable DAY in a task, will result in an error, for instance:

```
proc print data = invoices_totals;
   by day;
run;
```

```
ERROR: Data set WORK.INVOICES_TOTALS is not sorted in ascending sequence. The current BY group
       has day = Wednesday and the next BY group has day = Friday.
```

You can specify the NOTSORTED option in the BY statement to indicate that the data are not sorted in alphabetical order, but that they are grouped.

```
proc print data = invoices_totals;
   by day notsorted;
run;
```

SORT SEQUENCES

The internal presentation of a character on a z/OS machine is different from the presentation on a PC or Unix machine. The z/OS machine uses EBCDIC, whereas the Unix and PC platform use ASCII. This also impacts the collating sequence in a PROC SORT output. For example, numeric values in EBCDIC are presented by the highest hex values (F0 = 0, F1 = 1, …, F9 = 9) whereas in ASCII the same numbers are presented by low hex values (30 = 0, 31 = 1, …, 39 = 9, 41 = A, 61 = a).

When using data, created on an EBCDIC platform, on an ASCII platform you have to take into account this different collating sequence, if you treat the data using BY group processing.

The sort sequence of characters in EBCDIC is: blank . < … a b c … z A B C … Z 0 1 … 9

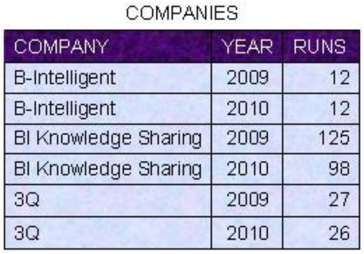
The sort sequence of these characters in ASCII is: blank . < 0 1 … 9 A B C … Z a b c … z

The SORT procedure by default uses the collating sequence of the platform on which the sort is executed. You can change this by specifying the SORTSEQ option.

Traditional Sort Sequences

Specifying the SORTSEQ option in the SORT or SQL procedure is very useful in client/server applications as illustrated by the following example.

Suppose that you prepare your data on a z/OS machine and that you transfer a sorted table to PC for further processing in BY groups. For instance, consider the following table, created on a z/OS machine, sorted with the – default – EBCDIC sequence.



COMPANIES

COMPANY	YEAR	RUNS
B-Intelligent	2009	12
B-Intelligent	2010	12
BI Knowledge Sharing	2009	125
BI Knowledge Sharing	2010	98
3Q	2009	27
3Q	2010	26

When this table is processed on PC in a procedure with a BY statement, the result is created, using the EBCDIC sequence, without any errors.

When you use this table in a DATA step MERGE statement, to combine the table with a PC table (sorted using the ASCII sequence), you will get an error:

```
ERROR: Input data sets cannot be combined because they have different collating sequences
       (SORTSEQ).
```

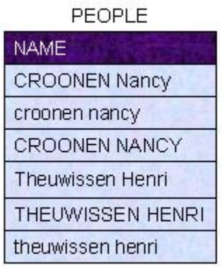
To solve this issue, avoiding an extra SORT step on PC, you can sort the data on the z/OS machine, using the SORTSEQ = ASCII option.

```
proc sort data = companies
          sortseq = ascii;
   by company;
run;
```

The SORTSEQ = LINGUISTIC Option

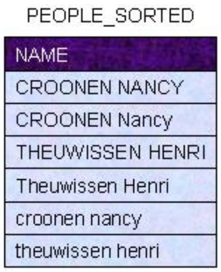
The LINGUISTIC sort sequence was introduced with SAS® 9.2, to enable more intuitive sorts. Consider the table PEOPLE, containing a variable NAME containing data in mixed case. Using a simple sort on this table will return an unwanted result. You expect a result that is not case sensitive.

```
proc sort data = people
          out = people_sorted;
   by name;
run;
```



PEOPLE

NAME
CROONEN Nancy
croonen nancy
CROONEN NANCY
Theuwissen Henri
THEUWISSEN HENRI
theuwissen henri

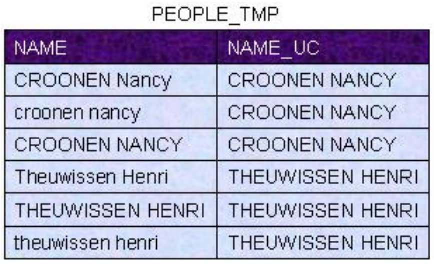


PEOPLE_SORTED

NAME
CROONEN NANCY
CROONEN Nancy
THEUWISSEN HENRI
Theuwissen Henri
croonen nancy
theuwissen henri

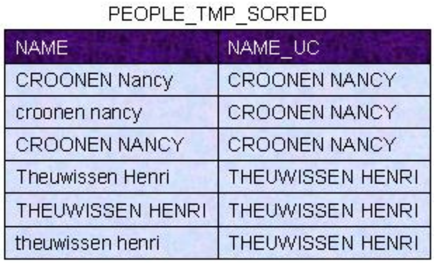
Before SAS® 9.2, the only way to solve this problem was to create a new variable, with all values in upper case or in lower case and sort on that new variable.

```
data people_tmp;
   set people;
   name_uc = upcase (name);
run;

proc sort data = people_tmp
          out = people_tmp_sorted;
   by name_uc;
run;
```



PEOPLE_TMP

NAME	NAME_UC
CROONEN Nancy	CROONEN NANCY
croonen nancy	CROONEN NANCY
CROONEN NANCY	CROONEN NANCY
Theuwissen Henri	THEUWISSEN HENRI
THEUWISSEN HENRI	THEUWISSEN HENRI
theuwissen henri	THEUWISSEN HENRI

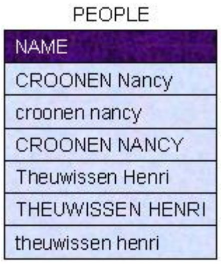


PEOPLE_TMP_SORTED

NAME	NAME_UC
CROONEN Nancy	CROONEN NANCY
croonen nancy	CROONEN NANCY
CROONEN NANCY	CROONEN NANCY
Theuwissen Henri	THEUWISSEN HENRI
THEUWISSEN HENRI	THEUWISSEN HENRI
theuwissen henri	THEUWISSEN HENRI

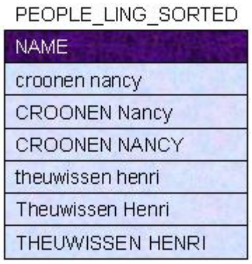
Specify the SORTSEQ = LINGUISTIC option in the PROC statement to request an intuitive accepted sort, following the locale used in the SAS session. The sort is not case sensitive.

```
proc sort data = people
          out = people_ling_sorted
          sortseq = linguistic;
   by name;
run;
```



PEOPLE

NAME
CROONEN Nancy
croonen nancy
CROONEN NANCY
Theuwissen Henri
THEUWISSEN HENRI
theuwissen henri



PEOPLE_LING_SORTED

NAME
croonen nancy
CROONEN Nancy
CROONEN NANCY
theuwissen henri
Theuwissen Henri
THEUWISSEN HENRI

The SORTSEQ = LINGUISTIC (ALTERNATE_HANDLING = SHIFTED) Option

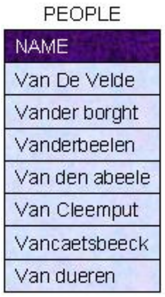
The SORTSEQ = LINGUISTIC option in the PROC statement allows additional parameters. The general syntax is shown below:

```
PROC SORT DATA = table-name SORTSEQ = LINGUISTIC (collating-rule = value);
```

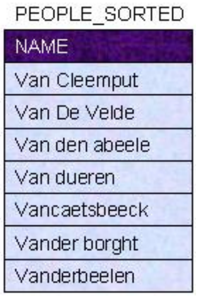
Variables that contain for instance name information often contain embedded blanks. Usually that data must be sorted, ignoring the blanks to get a result as in telephone lists.

Consider a table PEOLPE, with a variable NAME, containing information with embedded blanks. Using a simple sort on this table will return an unwanted result, caused by the embedded blanks.

```
proc sort data = people
          out = people_sorted;
   by name;
run;
```


PEOPLE

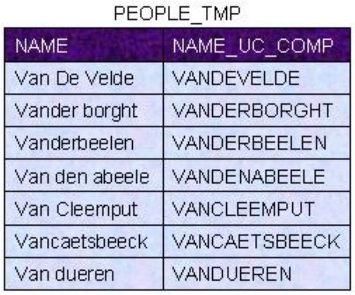
NAME
Van De Velde
Vander borght
Vanderbeelen
Van den abeele
Van Cleemput
Vancaetsbeeck
Van dueren


PEOPLE_SORTED

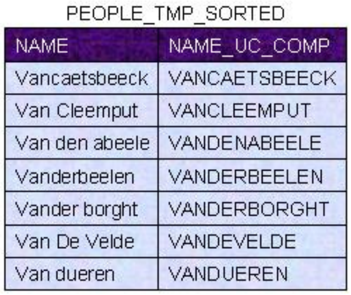
NAME
Van Cleemput
Van De Velde
Van den abeele
Van dueren
Vancaetsbeeck
Vander borght
Vanderbeelen

Prior to SAS® 9.2, the only way to solve this problem was to create a new variable, with all values in upper case or in lower case, and all embedded blanks removed, followed by a sort on that new variable.

```
data people_tmp;
   set people;
   name_uc_comp = upcase (compress (name));
run;

proc sort data = people_tmp
          out = people_tmp_sorted;
   by name_uc_comp;
run;
```


PEOPLE_TMP

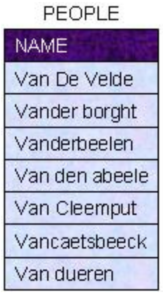
NAME	NAME_UC_COMP
Van De Velde	VANDEVELDE
Vander borght	VANDERBORGHT
Vanderbeelen	VANDERBEELEN
Van den abeele	VANDENABEELE
Van Cleemput	VANCLEEMPUT
Vancaetsbeeck	VANCAETSBEECK
Van dueren	VANDUEREN


PEOPLE_TMP_SORTED

NAME	NAME_UC_COMP
Vancaetsbeeck	VANCAETSBEECK
Van Cleemput	VANCLEEMPUT
Van den abeele	VANDENABEELE
Vanderbeelen	VANDERBEELEN
Vander borght	VANDERBORGHT
Van De Velde	VANDEVELDE
Van dueren	VANDUEREN

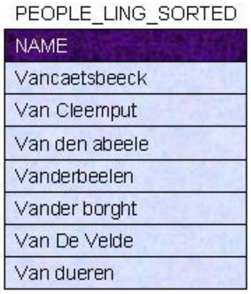
Use the collating rule ALTERNATE_HANDLING = SHIFTED to treat spaces as minimally important in the sort.

```
proc sort data = people
          out = people_ling_sorted
          sortseq = linguistic (alternate_handling = shifted);
   by name;
run;
```



PEOPLE

NAME
Van De Velde
Vander borght
Vanderbeelen
Van den abeele
Van Cleemput
Vancaetsbeeck
Van dueren



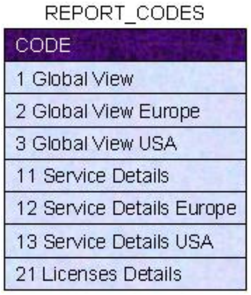
PEOPLE_LING_SORTED

NAME
Vancaetsbeeck
Van Cleemput
Van den abeele
Vanderbeelen
Vander borght
Van De Velde
Van dueren

The SORTSEQ = LINGUISTIC (NUMERIC_COLLATION = ON) Option

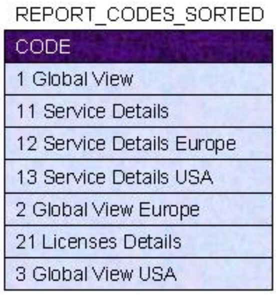
You often receive data, where a character variable starts with a number, indicating a specific sequence for the variable. Sorting on such a variable can be cumbersome. Consider the table REPORT_CODES, containing a variable CODE. Using a simple sort on this table will return an unwanted result.

```
proc sort data = report_codes
          out = report_codes_sorted;
   by code;
run;
```



REPORT_CODES

CODE
1 Global View
2 Global View Europe
3 Global View USA
11 Service Details
12 Service Details Europe
13 Service Details USA
21 Licenses Details

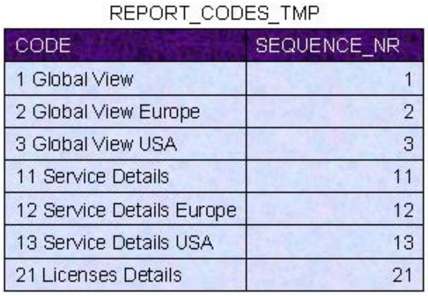


REPORT_CODES_SORTED

CODE
1 Global View
11 Service Details
12 Service Details Europe
13 Service Details USA
2 Global View Europe
21 Licenses Details
3 Global View USA

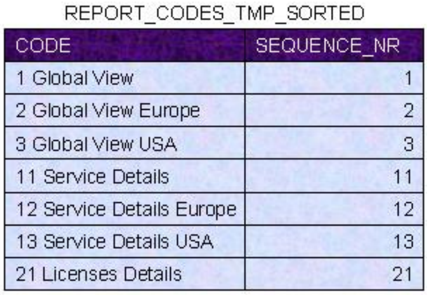
To build a sorted result, taking the numeric values at the beginning of the string as real numeric numbers, you can create a new numeric variable and then sort on that new variable.

```
data report_codes_tmp;
   set report_codes;
   length sequence_nr 4;
   sequence_nr = input (scan (code, 1, ' '), 4.);
run;

proc sort data = report_codes_tmp
          out = report_codes_sorted;
   by sequence_nr;
run;
```



REPORT_CODES_TMP

CODE	SEQUENCE_NR
1 Global View	1
2 Global View Europe	2
3 Global View USA	3
11 Service Details	11
12 Service Details Europe	12
13 Service Details USA	13
21 Licenses Details	21

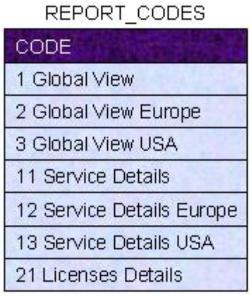


REPORT_CODES_TMP_SORTED

CODE	SEQUENCE_NR
1 Global View	1
2 Global View Europe	2
3 Global View USA	3
11 Service Details	11
12 Service Details Europe	12
13 Service Details USA	13
21 Licenses Details	21

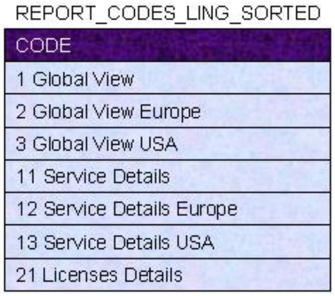
The collating rule NUMERIC_COLLATION = ON treats integer values within a string as their numeric equivalent for sorting.

```
proc sort data = report_codes
          out = report_codes_ling_sorted
          sortseq = linguistic (numeric_collation = on);
   by code;
run;
```



REPORT_CODES

CODE
1 Global View
2 Global View Europe
3 Global View USA
11 Service Details
12 Service Details Europe
13 Service Details USA
21 Licenses Details



REPORT_CODES_LING_SORTED

CODE
1 Global View
2 Global View Europe
3 Global View USA
11 Service Details
12 Service Details Europe
13 Service Details USA
21 Licenses Details

CONCLUSION

It takes 2 minutes to write a PROC SORT step, but it might take hours to execute that step! Most applications waste a lot of resources (CPU time, memory, disk space) due to redundant sorts or inefficient sorts. Often these applications can be simplified a lot by just removing several steps.

The better you know your data, the better you will be able to write efficient SAS code.

CONTACT INFORMATION

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

Name: Henri THEUWISSEN
Enterprise: BI Knowledge Sharing
Address: Sterrenlaan 40
B-3360 BIERBEEK
BELGIUM
Work Phone: +32 (0)496 28 45 28
E-mail: henri.theuwissen@biknowledgesharing.be
Web: www.BIKnowledgeSharing.be