INTRODUCTION

How often did you get a time ABEND in your SAS® job? How often did you get an “Out of Memory” error during your SAS job? How often did you get the error “Insufficient Space to store data set”? Analyzing vast quantities of data can not only be a chore but it can also be a resource hog.

This paper focuses on processing large amounts of data and explores the ways in which you can save CPU time, disk space and memory in your SAS jobs. Topics include:

- Basic efficiency techniques: things you know, but never use.
- Advanced utilization of formats.
- Save CPU time by avoiding sorting your data.
- Space leaks.
- The power of indexes.
- The danger of compression.
- Combining data.

All topics are based on real life experiences, discovered during consulting activities, mainly in an MVS environment. Most items are also applicable to other platforms.

Benchmarking results are summarized with tables whenever practical. Attendees should have a solid experience with Base SAS®.

SELECTING OBSERVATIONS

When you want to test for different values of a variable using the subsetting IF statement, you can choose between the IN operator or the OR operator. Intuitively you do not expect any difference in CPU time selecting either one of these methods. The examples below show that the IN operator requires more CPU time. The difference becomes even more important when testing on more values.

<table>
<thead>
<tr>
<th>PROGRAM NUMBER</th>
<th>DESCRIPTION</th>
<th>CPU TIME (SECONDS)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1-A</td>
<td>5 VALUES - IN</td>
<td>1.94</td>
</tr>
<tr>
<td>1-B</td>
<td>5 VALUES - OR</td>
<td>0.80</td>
</tr>
<tr>
<td>1-C</td>
<td>15 VALUES - IN</td>
<td>3.92</td>
</tr>
<tr>
<td>1-D</td>
<td>15 VALUES - OR</td>
<td>0.90</td>
</tr>
</tbody>
</table>

Subsetting data in a DATA step is possible through the IF statement or the WHERE statement. Usually the WHERE statement is more efficient than the IF statement, because the IF statement is executed on the data, being in the Program Data Vector, whereas the WHERE statement is executed before bringing the data in the Program Data Vector. The following examples show this behavior.
You need to be careful though when using functions in WHERE statements. The following examples show that using the SUBSTR function in a WHERE statement increases the CPU time incredibly compared to the corresponding IF statement. When using a typical WHERE operand (LIKE), the same subset is created, but CPU time decreases and gives a better performance again compared to the subsetting IF statement.

**Program 2-C**

```
DATA CLIENT;
  SET SUGI28.CLIENT;
  IF SUBSTR (LAST_NAME, 1, 3) = 'VAN';
RUN;
```

**Program 2-D**

```
DATA CLIENT;
  SET SUGI28CLIENT;
  WHERE SUBSTR (LAST_NAME, 1, 3) = 'VAN';
RUN;
```

**Program 2-E**

```
DATA CLIENT;
  SET SUGI28.CLIENT;
  WHERE LAST_NAME LIKE 'VAN%';
RUN;
```

<table>
<thead>
<tr>
<th>PROGRAM NUMBER</th>
<th>DESCRIPTION</th>
<th>CPU TIME (SECONDS)</th>
</tr>
</thead>
<tbody>
<tr>
<td>2-A</td>
<td>IF</td>
<td>0.09</td>
</tr>
<tr>
<td>2-B</td>
<td>WHERE</td>
<td>0.07</td>
</tr>
<tr>
<td>2-C</td>
<td>IF - SUBSTR</td>
<td>0.11</td>
</tr>
<tr>
<td>2-D</td>
<td>WHERE - SUBSTR</td>
<td>0.22</td>
</tr>
<tr>
<td>2-E</td>
<td>WHERE - LIKE</td>
<td>0.09</td>
</tr>
</tbody>
</table>

**Renaming Variables**

Combining tables using a MERGE statement in a DATA step requires a variable with the same name in both tables, to be used in the BY statement. When you need to combine two tables, and the common key field has a different name, many users write an additional DATA step, just to create a variable with the same name. It takes only a few seconds programmer time to write the DATA step, but this DATA step might require a lot of CPU time. Replacing this DATA step by the RENAME = option will save this CPU time.

**Program 1-A**

```
DATA SALES (DROP = CUSTOMER_ID);
  SET SUGI28.SALES;
  CLIENT_ID = CUSTOMER_ID;
RUN;
```

```
DATA SUGI28.CLIENTSALES;
  MERGE SUGI28.CLIENT
    SUGI28.SALES
    (RENAME = (CUSTOMER_ID = CLIENT_ID)
     IN = IN_SALES);
  BY CLIENT_ID;
  IF IN_SALES;
RUN;
```

**Program 1-B**

```
DATA SUGI28.CLIENTSALES;
  MERGE SUGI28.CLIENT
    SUGI28.SALES
    (RENAME = (CUSTOMER_ID = CLIENT_ID)
     IN = IN_SALES);
  BY CLIENT_ID;
  IF IN_SALES;
RUN;
```

<table>
<thead>
<tr>
<th>PROGRAM NUMBER</th>
<th>DESCRIPTION</th>
<th>CPU TIME (SECONDS)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1-A</td>
<td>NEW VARIABLE</td>
<td>5.66</td>
</tr>
<tr>
<td>1-B</td>
<td>RENAME</td>
<td>4.03</td>
</tr>
</tbody>
</table>

**Reducing Observation Length**

SAS® provides an entire library of powerful functions for data manipulation. Several of these functions have ‘space leaks’: If you do not specify a LENGTH statement to identify the resulting variable, you might waste a lot of disk space. Two examples illustrate this behavior.

Within the first example the variable INITIALS contains the values that you expect, but the length of this variable equals the sum of the contributing variables. As a result, every observation in the output table contains 38 redundant blanks. Taking the client table with 100.000 observations you waste 3.8 MB.
Some functions — like the SCAN function — create a result with a default length of 200, being the maximum length of a character variable in release 6.12 of SAS and in all earlier releases. Fortunately in release 8, this value remains 200, and is not increased to the maximum length of a character variable (32 K).

**PROGRAM 2-A**

```sas
DATA CLIENT;
  SET SUGI28.CLIENT;
  CARD_COUNTRY = SCAN (CLIENT_ID, 1, '-');
  CARD_CITY    = SCAN (CLIENT_ID, 2, '-');
  CARD_NUMBER  = SCAN (CLIENT_ID, 3, '-');
RUN;
```

**PROGRAM 2-B**

```sas
DATA CLIENT;
  SET SUGI28.CLIENT;
  LENGTH CARD_COUNTRY CARD_CITY $ 2
                     CARD_NUMBER $ 8;
  CARD_COUNTRY = SCAN (CLIENT_ID, 1, '-');
  CARD_CITY    = SCAN (CLIENT_ID, 2, '-');
  CARD_NUMBER  = SCAN (CLIENT_ID, 3, '-');
RUN;
```

### FORMATTING

Formats are often used to group items together, without creating an additional variable. Did you ever consider working on the length of a formatted value to add an additional level of aggregation?

**PROGRAM 1-A**

```sas
DATA SALES;
  SET SUGI28.SALES;
  LENGTH PRODUCT_GROUP $ 1;
  PRODUCT_GROUP = SUBSTR (PRODUCT_ID, 1, 1);
RUN;
```

```sas
PROC TABULATE DATA = SUGI28.SALES
  FORMAT = COMMA12. NOSEPS;
  CLASS PRODUCT_GROUP;
  FORMAT PRODUCT_ID $1.;
  VAR SALES_AMOUNT;
  TABLE PRODUCT_ID = ' ' ALL = 'TOTAL',
                    SALES_AMOUNT * SUM = ' '
                    / RTS = 15 BOX = 'PRODUCT GROUP';
RUN;
```

**PROGRAM 2-A**

```sas
DATA SALES;
  SET SUGI28.SALES;
  SALES_YEAR = YEAR (SALES_DATE);
RUN;
```

```sas
PROC TABULATE DATA = SALES
  FORMAT = COMMA12. NOSEPS;
  CLASS SALES_YEAR;
  VAR SALES_AMOUNT;
  TABLE SALES_YEAR = ' ' ALL = 'TOTAL',
                    SALES_AMOUNT * SUM = ' '
                    / RTS = 12 BOX = 'SALES YEAR';
RUN;
```

### PROGRAMS

<table>
<thead>
<tr>
<th>PROGRAM NUMBER</th>
<th>DESCRIPTION</th>
<th>ADDITIONAL LENGTH</th>
</tr>
</thead>
<tbody>
<tr>
<td>1-A</td>
<td>SUBSTR</td>
<td>20 + 20 = 40</td>
</tr>
<tr>
<td>1-B</td>
<td>SUBSTR - LENGTH</td>
<td>2</td>
</tr>
<tr>
<td>2-A</td>
<td>SCAN</td>
<td>3 x 200 = 600</td>
</tr>
<tr>
<td>2-B</td>
<td>SCAN - LENGTH</td>
<td>2 + 2 + 8 = 12</td>
</tr>
</tbody>
</table>

Within a large SAS program you might have several steps containing the same subsetting conditions. Maintenance of these programs often generates errors: you forget to modify the subsetting condition in one or more steps. To facilitate the maintenance process, you might create a user-defined format, and apply this format in the subsetting conditions, instead of specifying a large list of values. When changes are required, it is sufficient to update the user-defined format without having to adjust the subsetting condition at several places.

**PROGRAM 3-A**

```sas
DATA SALES_BELGIUM;
  SET SUGI28.SALES;
  WHERE SHOP_ID = 'B-A' OR
         SHOP_ID = 'B-B' OR
         SHOP_ID = 'B-L' OR
         SHOP_ID = 'B-G';
RUN;
```
SORTING

Very often data is required in a specific order. It only takes a few seconds programmer time to type a PROC SORT step, and you often find too many PROC SORT steps in SAS programs.

Through the SORT flag, stored in the descriptor portion of a SAS data set, a lot of redundant sorts are avoided. The SORT flag is copied to another table if a sorted table is treated by a procedure like for example the COPY procedure. A DATA step though will not copy the SORT flag, since within a DATA step you might add observations, change the value of the variable defined in the SORT flag, etc.

Several users are not familiar with the SORTEDBY = option that was introduced in SAS Release 6.

PROGRAM 3-B

PROC FORMAT LIB = SUGI28 FMTLIB;
   VALUE $SHOPFMT 'B-A',
      'B-B',
      'B-L',
      'B-G' = 'BELGIUM'
      'NL-A',
      'NL-DH',
      'NL-R' = 'THE NETHERLANDS';
RUN;

OPTIONS FMTSEARCH = (SUGI28);

DATA SALES_BELGIUM;
   SET SUGI28.SALES;
   WHERE PUT (SHOP_ID, $SHOPFMT.) = 'BELGIUM';
RUN;

<table>
<thead>
<tr>
<th>PROGRAM NUMBER</th>
<th>ADDITIONAL MAINTENANCE</th>
</tr>
</thead>
<tbody>
<tr>
<td>3-A</td>
<td>FOR EACH USAGE (WHERE)</td>
</tr>
<tr>
<td>3-B</td>
<td>ONCE (FORMAT)</td>
</tr>
</tbody>
</table>

Several consolidating procedures like TABULATE, FREQ, MEANS, SUMMARY return the result in sorted order. Still many users precede these procedures by a PROC SORT, sorting on the variables specified in the CLASS statement. According to these users, the execution of the subsequent procedure will be
faster. They forget to examine that they require more CPU time to sort the data than the gain in the subsequent procedure.

The only reason to have a SORT procedure preceding for example the MEANS procedure is that you need BY processing.

**PROGRAM 2-A**

PROC SORT DATA = SUGI28.SALES OUT = SALES;
   BY SHOP_ID CUSTOMER_ID;
RUN;

PROC SUMMARY DATA = SALES NWAY;
   CLASS SHOP_ID CUSTOMER_ID;
   VAR SALES_AMOUNT;
   OUTPUT OUT = SUMSALES SUM =;
RUN;

**PROGRAM 2-B**

PROC SUMMARY DATA = SUGI28.SALES NWAY;
   CLASS SHOP_ID CUSTOMER_ID;
   VAR SALES_AMOUNT;
   OUTPUT OUT = SUMSALES SUM =;
RUN;

<table>
<thead>
<tr>
<th>PROGRAM NUMBER</th>
<th>DESCRIPTION</th>
<th>CPU TIME (SECONDS)</th>
</tr>
</thead>
<tbody>
<tr>
<td>2-A</td>
<td>SORT - SUMMARY</td>
<td>10.50</td>
</tr>
<tr>
<td>2-B</td>
<td>SUMMARY</td>
<td>4.80</td>
</tr>
</tbody>
</table>

**INDEXING**

Although an index is considered for use in a WHERE statement and not in a subsetting IF statement, you still find several programs using an IF statement to subset a table with an index. The gain in CPU time becomes more important if the subset returned by the index is smaller. In the following examples, a simple index exists on the variables SHOP_ID and CUSTOMER_ID. The variable SHOP_ID has only 7 distinct values, whereas the variable CUSTOMER_ID contains approximately 80,000 different values.

Accessing the data through the index on SHOP_ID returns +/- 15% of the data, resulting in only a small difference between the WHERE statement (using the index) and the IF statement (performing a sequential search).

**PROGRAM 1-A**

DATA SALES_B_B;
   SET SUGI28.SALES_INDEXED;
   IF SHOP_ID = 'B-B';
RUN;

**PROGRAM 1-B**

DATA SALES_B_B;
   SET SUGI28.SALES_INDEXED;
   WHERE SHOP_ID = 'B-B';
RUN;

**PROGRAM 1-C**

DATA SALES_B_B;
   SET SUGI28.SALES_INDEXED;
   WHERE SHOP_ID = 'B-B';
RUN;

Accessing the data through the index on CUSTOMER_ID returns less than 0.01% of the data and is extremely fast compared to the subsetting IF statement.

**PROGRAM 2-A**

DATA SALES_NL_A_31678197;
   SET SUGI28.SALES_INDEXED;
   IF CUSTOMER_ID = 'NL-A-31678197';
RUN;

**PROGRAM 2-B**

DATA SALES_NL_A_31678197;
   SET SUGI28.SALES_INDEXED;
   WHERE CUSTOMER_ID = 'NL-A-31678197';
RUN;

<table>
<thead>
<tr>
<th>PROGRAM NUMBER</th>
<th>DESCRIPTION</th>
<th>CPU TIME (SECONDS)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1-A</td>
<td>7 SHOPS - IF</td>
<td>1.31</td>
</tr>
<tr>
<td>1-B</td>
<td>7 SHOPS - WHERE</td>
<td>1.02</td>
</tr>
<tr>
<td>2-A</td>
<td>100.000 CLIENTS - IF</td>
<td>0.76</td>
</tr>
<tr>
<td>2-B</td>
<td>100.000 CLIENTS - WHERE</td>
<td>0.01</td>
</tr>
</tbody>
</table>

**COMPRESSING**

Compression can be useful if disk space is a problem. Compression must be added in a sensible way: Both compressing the data and decompressing the data requires CPU time.

Never specify the COMPRESS = YES option in the global OPTIONS statement.

The following examples illustrate the CPU cost of compression: an input SAS data set is sorted into an output SAS data set. All combinations of compressed - not compressed are examined.

**PROGRAM 1-A**

PROC SORT DATA = SUGI28.CLIENT
   OUT = CLIENT;
   BY HOME_CITY;
RUN;

**PROGRAM 1-B**

PROC SORT DATA = SUGI28.CLIENT
   OUT = CLIENT_COMPRESSED
      (COMPRESS = YES);
   BY HOME_CITY;
RUN;

**PROGRAM 1-C**

PROC SORT DATA = SUGI28.CLIENT_COMPRESSED
   OUT = CLIENT;
   BY HOME_CITY;
RUN;
SUBSETTING EXTERNAL FILES

The INPUT statement, structuring the input buffer's content into variables in the Program Data Vector will consume quite some CPU time. If you only need to process a subset of the external file, only examine part of the input buffer, and if this part meets your subsetting condition, examine the rest of the input buffer. The trailing @ in the INPUT statement allows holding contents the input buffer.

### PROGRAM 1-A

```sas
DATA CLIENT;
  INFILE CLIENT;
  INPUT CLIENT_ID $ 1 - 14
  LAST_NAME $ 16 - 35
  FIRST_NAME $ 37 - 56
  HOME_CITY $ 58 - 77
  HOME_COUNTRY $ 79 - 93
  ...;
RUN;

DATA CLIENT_LEUVEN;
  SET CLIENT;
  IF HOME_CITY = 'LEUVEN';
RUN;
```

### PROGRAM 1-B

```sas
DATA CLIENT_LEUVEN;
  INFILE CLIENT;
  INPUT CLIENT_ID $ 1 - 14
  LAST_NAME $ 16 - 35
  FIRST_NAME $ 37 - 56
  HOME_CITY $ 58 - 77
  HOME_COUNTRY $ 79 - 93
  ...;
  IF HOME_CITY = 'LEUVEN';
RUN;
```

### PROGRAM 1-C

```sas
DATA CLIENT_LEUVEN;
  INFILE CLIENT;
  INPUT HOME_CITY $ 58 - 77 @;
  IF HOME_CITY = 'LEUVEN';
  INPUT CLIENT_ID $ 1 - 14
  LAST_NAME $ 16 - 35
  FIRST_NAME $ 37 - 56
  HOME_COUNTRY $ 79 - 93
  ...;
RUN;
```

The CPU gain will be more important when processing more input lines and when the input record creates more variables.

### EFFICIENTLY SUMMARIZING DATA

#### CREATING SUMMARIZED SAS DATA SETS

Sometimes one column uniquely identifies other columns in a table. In other words, there is a 1-to-1 relation between this unique identifier and several other columns in the table. For example, a client ID uniquely identifies 1 client name, who lives in 1 city, etc.

Suppose you need to consolidate data, and the consolidated result should contain the unique identifier as well as some other descriptive information. In that case, you should consider using an ID statement instead of specifying all columns in the CLASS statement. There is an important gain in memory usage and CPU time.

### PROGRAM 1-A

```sas
PROC SUMMARY DATA = SUGI28.CLIENTSALES;
CLASS CLIENT_ID
  LAST_NAME
  FIRST_NAME
  HOME_CITY
  HOME_COUNTRY;
VAR SALES_AMOUNT;
OUTPUT OUT = SUMSALES SUM =;
RUN;
```

### PROGRAM 1-B

```sas
PROC SUMMARY DATA = SUGI28.CLIENTSALES;
CLASS CLIENT_ID;
ID LAST_NAME
  FIRST_NAME
  HOME_CITY
  HOME_COUNTRY;
VAR SALES_AMOUNT;
OUTPUT OUT = SUMSALES SUM =;
RUN;
```
The **SUMMARY** procedure by default creates an output SAS data set with all the different _TYPE_ values. If you are interested in only a few of them, there are several possibilities to create this subset, as shown in the following examples:

- Very often users execute the **SUMMARY** procedure, followed by an additional DATA step to remove the redundant _TYPE_ observations. Most users never code a WHERE = option for an output SAS data set. This will remove the need of the extra DATA step, and even your **SUMMARY** procedure will be faster, since less records have to be written to the output SAS data set.

- With release 8 SAS introduced new statements to request only specific combinations of CLASS variables. The TYPES statement identifies which of the possible combinations of CLASS variables to generate. The WAYS statement specifies the number of ways to make unique combinations of CLASS variables. These statements will reduce the memory requirements, but consume more CPU time.

- Finally an attempt is made to execute the consolidation in 2 phases: First a **SUMMARY** procedure is executed, creating the NWAY result. Afterwards a second **SUMMARY** procedure is executed on this (small) NWAY table.

**PROGRAM 2-A**

```
PROC SUMMARY DATA = SUGI28.CLIENTSALES;
   CLASS SHOP_ID
       PRODUCT_ID
       HOME_CITY
       LANGUAGE;
   VAR SALES_AMOUNT;
   OUTPUT OUT = SUMSALES SUM =;
RUN;
```

DATA SUMSALES;
   SET SUMSALES;
   IF _TYPE_ IN (3, 5, 6, 9, 10, 12);
RUN;

**PROGRAM 2-B**

```
PROC SUMMARY DATA = SUGI28.CLIENTSALES;
   CLASS SHOP_ID
       PRODUCT_ID
       HOME_CITY
       LANGUAGE;
   VAR SALES_AMOUNT;
   OUTPUT OUT = SUMSALES
      (WHERE = (_TYPE_ IN (3, 5, 6, 9, 10, 12))) SUM =;
RUN;
```

**PROGRAM 2-C**

```
PROC SUMMARY DATA = SUGI28.CLIENTSALES;
   CLASS SHOP_ID
       PRODUCT_ID
       HOME_CITY
       LANGUAGE;
   VAR SALES_AMOUNT;
   WAYS 2;
   OUTPUT OUT = SUMSALES SUM =;
RUN;
```

**PROGRAM 2-D**

```
PROC SUMMARY DATA = SUGI28.CLIENTSALES;
   CLASS SHOP_ID
       PRODUCT_ID
       HOME_CITY
       LANGUAGE;
   VAR SALES_AMOUNT;
   TYPES SHOP_ID * PRODUCT_ID
       SHOP_ID * HOME_CITY
       SHOP_ID * LANGUAGE
       PRODUCT_ID * HOME_CITY
       PRODUCT_ID * LANGUAGE
       HOME_CITY * LANGUAGE;
   OUTPUT OUT = SUMSALES SUM =;
RUN;
```
CREATING SUMMARY REPORTS

The ID statement, discussed with the SUMMARY procedure is not available in the TABULATE procedure. Specifying a lot of variables in the CLASS statement and as a crossing in the TABLE statement will result in ‘Out of Memory’ errors and high CPU values.

To create the result without memory problems, using less CPU time, precede the TABULATE procedure with a DATA step or DATA step VIEW that creates 1 variable, concatenating the other variables, and use this new variable in the TABLE statement in the TABULATE procedure.

PROGRAM 1-A

PROC TABULATE DATA = SUGI28.CLIENTSALES
   FORMAT = COMMA12. NOSEPS;
   CLASS CLIENT_ID LAST_NAME FIRST_NAME;
   VAR SALES_AMOUNT;
   TABLE CLIENT_ID * LAST_NAME * FIRST_NAME,
      SALES_AMOUNT / RTS = 60;
RUN;

PROGRAM 1-B

DATA CLIENTSALES (DROP = CLIENT_ID
   FIRST_NAME);
SET SUGI28.CLIENTSALES;
CLIENT = CLIENT_ID !
   LAST_NAME !
   FIRST_NAME;
RUN;
PROC TABULATE DATA = CLIENTSALES
   FORMAT = COMMA12. NOSEPS;
   CLASS CLIENT;
   VAR SALES_AMOUNT;
   TABLE CLIENT, SALES_AMOUNT / RTS = 60;
RUN;

EFFICIENTLY COMBINING DATA

CONCATENATING SAS DATA SETS

Many users are familiar with the APPEND procedure for adding a new table immediately to a master table, without reading / writing the master table. Still, they rarely code the APPEND procedure, because they are used to typing the DATA step, which is coded very fast.

In the next example the traditional DATA step concatenation capabilities are compared with using the OUTER UNION CORR operator in the SQL procedure. The result can also be created using the SQL INSERT statement to add all observations of the second table to the end of the master table.

PROGRAM 1-A

DATA SALES;
   SET SUGI28.SALES_B SUGI28.SALES_NL;
RUN;
PROC SORT DATA = SALES;
   BY SALES_DATE;
RUN;

PROGRAM 1-B

DATA SALES;
   SET SUGI28.SALES_B SUGI28.SALES_NL;
RUN;
PROC SQL;
   CREATE TABLE SALES AS
   SELECT * FROM SUGI28.SALES2003;
QUIT;

PROGRAM 1-C

PROC APPEND BASE = SALES
   DATA = SUGI28.SALES2003;
RUN;

PROGRAM 1-D

PROC SQL;
   INSERT INTO SALES
      SELECT * FROM SUGI28.SALES2003;
QUIT;

INTERLEAVING SAS DATA SETS

You can concatenate two sorted input SAS data sets into a sorted result in several ways. The following example compares the traditional DATA step followed by a SORT procedure with a BY statement immediately specified in the DATA step and with the OUTER UNION CORR operator with an ORDER BY clause in the SQL procedure. As expected the SQL procedure requires more CPU time than the DATA step.

PROGRAM 1-A

DATA SALES;
   SET SUGI28.SALES_B SUGI28.SALES_NL;
RUN;
PROC SORT DATA = SALES;
   BY SALES_DATE;
RUN;

PROGRAM 1-B

DATA SALES;
   SET SUGI28.SALES_B SUGI28.SALES_NL;
   BY SALES_DATE;
RUN;
MATCH-MERGING SAS DATA SETS

When a table is the result of combining and consolidating three different input tables without a common key field, coding an SQL procedure is probably the most easy solution. It requires though more CPU time than a combination of several PROC SORT steps and DATA steps. The following examples illustrate the difference between SQL and traditional DATA step processing. A comparison between using the DISTINCT keyword or multiple variables in the GROUP BY clause in the SQL procedure is also examined.

PROGRAM 1-A

PROC SORT DATA = SUGI28.PURCHASE
(KEEP = CUSTOMER_ID
PRODUCT_ID
QUANTITY)
OUT = PURCHASE;
BY PRODUCT_ID;
RUN;

DATA PRODUCTSALES (KEEP = CUSTOMER_ID
PRODUCT_ID
QUANTITY);
MERGE PURCHASE (IN = IN_PURCHASE)
SUGI28.PRODUCT (KEEP = PRODUCT_ID
UNIT_PRICE);
BY PRODUCT_ID;
IF IN_PURCHASE;
PRODUCTSALES = QUANTITY * UNIT_PRICE;
RUN;

PROC SORT DATA = PRODUCTSALES
(RENAME = (CUSTOMER_ID = CLIENT_ID));
BY CLIENT_ID;
RUN;

DATA CLIENTSALES;
MERGE PRODUCTSALES (IN = IN_PRODUCTSALES)
SUGI28.CLIENT (KEEP = CLIENT_ID
LAST_NAME
FIRST_NAME);
BY CLIENT_ID;
IF IN_PRODUCTSALES;
IF FIRST.CLIENT_ID THEN CLIENTSALES = 0;
CLIENTSALES + PRODUCTSALES;
IF LAST.CLIENT_ID;
RUN;
QUIT;

PROGRAM 1-B

PROC SQL;
CREATE TABLE CLIENTSALES AS
SELECT DISTINCT CLIENT_ID,
LAST_NAME,
FIRST_NAME,
SUM (QUANTITY * UNIT_PRICE) AS CLIENTSALES
FROM SUGI28.CLIENT C,
SUGI28.PRODUCT PR,
SUGI28.PURCHASE PU
WHERE C.CLIENT_ID = PU.CUSTOMER_ID
AND
PR.PRODUCT_ID = PU.PRODUCT_ID
GROUP BY CLIENT_ID,
LAST_NAME,
FIRST_NAME
ORDER BY CLIENTSALES DESC;
QUIT;

PROGRAM 1-C

PROC SQL;
CREATE TABLE CLIENTSALES AS
SELECT CLIENT_ID,
LAST_NAME,
FIRST_NAME,
SUM (QUANTITY * UNIT_PRICE) AS CLIENTSALES
FROM SUGI28.CLIENT C,
SUGI28.PRODUCT PR,
SUGI28.PURCHASE PU
WHERE C.CLIENT_ID = PU.CUSTOMER_ID
AND
PR.PRODUCT_ID = PU.PRODUCT_ID
GROUP BY CLIENT_ID
ORDER BY CLIENTSALES DESC;
QUIT;
CONCLUSION

Many users write SAS programs to provide a quick solution for ad hoc questions. Since they assume that the programs will be executed only once no effort is spent on the efficiency of these jobs.

However, often these programs become part of production jobs without verifying their efficiency due to strict deadlines. As a result a lot of CPU time and disk space is wasted, sometimes even requiring an earlier upgrade of the hardware with an important financial impact.

In this paper we demonstrated that several techniques are available to reduce the resources (CPU time, disk space, memory usage) needed by your SAS jobs. It might just require a little more programmer time.

CONTACT INFORMATION

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

Nancy CROONEN
SOLID Partners NV
Minervastraat 14 bis
B-1930 ZAVENTEM
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

SAS and all other SAS Institute Inc. product or service names are registered trademarks or trademarks of SAS Institute Inc. in the USA and other countries. ® indicates USA registration.

Other brand and product names are trademarks of their respective companies.

<table>
<thead>
<tr>
<th>PROGRAM NUMBER</th>
<th>DESCRIPTION</th>
<th>CPU TIME (SECONDS)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1-A</td>
<td>SQL</td>
<td>5711.03</td>
</tr>
<tr>
<td>1-B</td>
<td>DATA</td>
<td>1912.00</td>
</tr>
</tbody>
</table>