Möglichkeiten der SAS Software für die Analyse großer Datentabellen

Hans-Peter Altenburg
Deutsches Krebsforschungszentrum
Klinische Epidemiologie / C0500
Im Neuenheimer Feld 280
D-69120 Heidelberg
hp.altenburg@dkfz.de

Zusammenfassung

Nicht nur in großen epidemiologischen Studien gibt es manchmal die Situation, dass sowohl viele Probanden als auch eine große Anzahl von Merkmalen erfasst wurden. Leicht kann bei ineffizienter Programmierung die Zeit bis die Ergebnisse einer Analyse vorliegen für einen ungeduldigen Anwender zu lang geraten – obwohl ein schneller PC benutzt wurde.
Im Paper soll diskutiert werden, welche Möglichkeiten das SAS-System bietet, wenn umfangreiche Datentabellen mit Hilfe von SAS analysiert werden sollen. Dabei soll nicht nur Wert auf eine effiziente Programmierung im DATA-Step, sondern auch die effektive (Aus-) Nutzung von Prozeduren aufgezeigt werden. Insbesondere soll auch beschrieben werden, welche Möglichkeiten bestehen CPU- bzw. Ein- oder Ausgabezeit zu sparen.

Keywords: System Performanz, `KEEP-` / `DROP`-Statement, `WHERE`-Statement, temporäre Arrays, `OBS=`, `FIRSTOBS=`, `CLASS`-Statement, Prozedur `SORT`, Prozedur `SQL`, `TAGSORT`-Option, Index.

1 Einführung

Im folgenden sollen ein paar allgemeine Möglichkeiten zur Optimierung der Speicherausnutzung und zur Optimierung der CPU-Performance bei SAS-Programmen aufgezeigt werden, wie sie insbesondere bei der Auswertung von

großen Datentabellen wichtig werden können. Im letzten Abschnitt wird auf das Sortieren mit Hilfe der Prozedur `SORT` und einige Sortier-Alternativen von Datentabellen eingegangen.

Zwei einführende Beispiele sollen in die Problematik einführen. In einer multizentrischen Kohortenstudie über den Zusammenhang zwischen Ernährungsverhalten und Krebsentstehung werden Risikofaktoren zur Entstehung von Lungenkrebs untersucht. Die Zahl der Probanden in einer Subkohorte betrug n=417.717 mit ca. 600 Variablen. Der verwendete PC war mit 128 MB RAM und einem 1Ghz-Prozessor ausgestattet. Im ersten Beispiel werden drei Varianten für eine einfache Häufigkeitsanalyse mit Hilfe der SAS-Prozedur `FREQ` vorgestellt, die eine unterschiedliche Systemperformanz erkennen lassen. Es werden jeweils die Statements im LOG-File angegeben.

Beispiel 1.1:
Lungenkrebsfälle sollen nach dem Geschlecht gegenüber gestellt werden:

```
1    %LET dset=lung.lyon061e  ;
2
3    PROC FREQ DATA=&dset  ;
4    TABLES
5           sex*caselung / nopercent
6            ;
7    RUN ;

NOTE: There were 417717 observations read from the data set
      LUNG.LYON061E.
NOTE: PROCEDURE FREQ used:
      real time           1:19.31
      cpu time            8.50 seconds
```

Beispiel 1.2:
Diese Variante verwendet die Datei-Option `KEEP`.

```
8
9    PROC FREQ DATA=&dset (KEEP=sex caselung) ;
10   TABLES
11          sex*caselung / nopercent
12           ;
13   RUN ;

NOTE: There were 417717 observations read from the data set
      LUNG.LYON061E.
NOTE: PROCEDURE FREQ used:
      real time           1:18.67
      cpu time            9.40 seconds
```

Beispiel 1.3:
Über ein SAS-Makro geht alles noch einen kleinen Tick schneller.

```
14
15   %MACRO tabs(row,col) ;
16   PROC FREQ DATA=&dset (KEEP=&row &col) ;
17   TABLES
18          (&row)*(&col) / NOPERCENT   ;
19   RUN ;
20   %MEND   ;
21
22   %tabs(sex,caselung)

NOTE: There were 417717 observations read from the data set
      LUNG.LYON061E.
NOTE: PROCEDURE FREQ used:
      real time           1:18.48
      cpu time            9.18 seconds
```

Beispiel 2:
Die Effektivitätssteigerung beim Verwenden der Datei-Option `KEEP` lässt sich leicht bei komplexeren SAS-Prozeduren erkennen. So sollten mit Hilfe des Cox-Proportional Hazard-Modells für eine bestimmte Modellsituation relative Risiken bestimmt werden. Hierzu wurde die SAS-Prozedur `PHREG` auf die Datenmenge des ersten Beispiels angewandt. Ohne das `KEEP`-Statement betrug die Zeitdauer (real time) mehr als zwei Stunden (Abbruch des Programms nach 2h), dagegen wurden mit einer `KEEP`-Datei-Option nur noch knapp 20 Minuten benötigt.

Schon diese Beispiele zeigen, dass es im SAS-System durchaus noch Möglichkeiten gibt, die Effektivität der Programmausführung zu erhöhen. Einige dieser Aspekte sollen im folgenden beschrieben werden.

2 Optimierung der System Performance

2.1 I/O Optimierung

Einer der wichtigsten Faktoren, um die Performanz zu erhöhen, ist die I/O-Optimierung. Da in der Regel ein Daten-Analyse-Prozess aus vielen Zyklen mit Lesen, Schreiben oder der Manipulation von Datentabellen befasst ist, kann die Programmperformance deutlich verbessert werden, wenn die Zahl der Plattenzugriffe verringert werden kann. Hieraus resultiert die Forderung ein SAS Programm so konzipieren, dass möglichst wenige, d.h. besser nur die erforder-

lichen Variablen und Beobachtungen verwendet werden. Die folgenden Tipps und Beispiele sollen helfen, die Anzahl der I/O-Operationen zu optimieren.

1. Lese nur die benötigten Felder:

```
DATA neu ;
INFILE 'D:\...\...\dsetraw.txt' ;
INPUT ID $ 1-10 G $ 11 @110 x1 10.0 x2 12.2 ;
RUN ;
```

   Mit Hilfe des Zeigers (Positionierzeichen) `'@'` werden die Informationen in den Spalten 12 bis 109 überlesen.

2. Speichere Daten in eine permanente SAS-Datei:

```
LIBNAME  st_0102 'G:\...\...\daten' ;
DATA st_0102.neu ;
INFILE 'D:\...\...\dsetraw.txt' ;
INPUT ID $ 1-10 G $ 11 @110 x1 10.0 x2 12.2 ;
RUN ;
```

   Die Daten werden eingelesen und gleich in eine permanente SAS-Datei (SAS-Datentabelle) abgespeichert. Über das `LIBNAME`-Statement wird die Verknüpfung zum Pfad des richtigen Ordners hergestellt.

3. Hilfen zur I/O Optimierung. Folgende Statements bzw. Datei-Optionen können helfen die Anzahl der Variablen in einer SAS-Datentabelle zu reduzieren:

```
KEEP bzw. DROP-Statement
WHERE-Ausdruck
OBS= / FIRSTOBS=
LENGTH
```

3.1 Reduktion der Variablenanzahl über die Verwendung des `KEEP`- bzw. `DROP`-Statements im DATA-Step. Die hinter dem Schlüsselwort `KEEP` aufgelisteten Variablen werden in der SAS-Datentabelle behalten. Das `DROP`-Statement hat die komplementäre Wirkung. Im folgenden Beispiel wird nur die Variable x in der SAS-Datentabelle behalten. Die Variable `sum_x` wird (wenn kein nachfolgendes „`KEEP sum_x ;`“ vorkommt) nicht behalten!

```
DATA st_0102.new ; SET old ;
KEEP x ;
     sum_x + x ;
     . . .
RUN ;
```

Diese Statements sind weniger effizient als die Verwendung der entsprechenden Dateioptionen, weil bei der Verwendung einer Datei-Option, die nicht benannten Variablen gar nicht erst eingelesen werden. Die folgende Datentabelle enthält deshalb zwei Variablen `x` sowie `sum_x`:

```
DATA st_0102.new ; SET old (KEEP=x );
sum_x + x ;
. . .
RUN ;
```

oder z.B. die Berechnung des arithmetischen Mittels:

```
DATA st_0102.new ;
     SET old (KEEP=x ) END=fin ;
sum_x + x ;       nn+1 ;
IF fin THEN mean_x=sum_x/nn ;
DROP sum_x nn ;
RUN ;
```

Über die Prozedur `PRINT` kann das Ergebnis ausgedruckt werden:

```
PROC PRINT DATA=new (KEEP=mean_x ) ;
RUN ;
```

und liefert die folgende **Ausgabe**:

```
Obs    mean_x

 1     .
 2     .
 3     .
 4     .
 5     .
 6     .
 7     .
 8     .
 9     .
10     .
11     .
12     .
13     .
14     .
15     .
16     .
17     .
18     .
19     .
20 0.45524
```

Die SAS-Datei `st_0102.new` enthält jetzt nur noch eine Variable, die bis auf die letzte Beobachtung mit fehlenden Werten belegt ist.

3.2 Reduktion der Anzahl von Beobachtungen unter Verwendung von `WHERE` zum Bilden einer Untermenge der Datei. Wie bereits `KEEP` bzw. `DROP` kann auch `WHERE` sowohl im Data-Step oder als Datei-Option verwendet werden.

Vorteil bei Verwendung als Datei-Option:
Der Zeitpunkt der Ausführung liegt vor dem Einlesen der Daten in den „program data vector“ (PDV), dies bringt vor allem Vorteile bei Dateien mit vielen Variablen! Außerdem erlaubt es weitere Operatoren wie z.B. `LIKE, CONTAINS` oder `=*` (phonetische Äquivalente) zu verwenden.

Nachteil:
Es kann nur einmal am Anfang verwendet werden, und es können keine temporären Variablen wie z.B. `_N_ / FIRST. / LAST. / IN` etc. benutzt werden.

Beispiele:
Als Dateioption:

```
PROC PRINT DATA=new (WHERE=(sex=1));
PROC PRINT DATA=new (WHERE=(30<=age<40));
```

Mit `WHERE`-Statement:

```
PROC PRINT DATA=new ; WHERE  sex=1 ;
PROC PRINT DATA=new ; WHERE=(30<=age<40) ;
```

Bildung einer Untermenge der Datei mit Hilfe einer `IF`-Bedingung:

```
DATA new ; SET old  END=fin ;
IF sex=1 THEN DO; sum_x + x ; nn+1 ; END ;
IF fin   THEN mean_x=sum_x/nn ;
DROP sum_x nn ;
RUN ;
```

Dto. über eine `WHERE`-Bedingung im DATA-Step:

```
DATA new ; SET old  END=fin ;
WHERE sex=1 ;
sum_x + x ; nn+1 ;
IF fin THEN mean_x=sum_x/nn ;
DROP sum_x nn ;
RUN ;
```

WHERE-Bedingung als DATA-Set-Option:

```
DATA new ; SET old (WHERE=(sex=1))  END=fin ;
sum_x + x ; nn+1 ;
IF fin THEN mean_x=sum_x/nn ;
DROP sum_x nn ;
RUN ;
```

Das obige Beispiel zur Berechnung des Mittelwerts in einer anderen Variante und einer deutlich verkürzten Ausgabe:

```
DATA new ; SET old (KEEP=x ) END=fin ;
sum_x + x ; nn+1;
IF fin THEN mean_x=sum_x/nn ;
DROP sum_x nn ;
RUN ;
PROC PRINT DATA=new (KEEP=mean_x  WHERE=(mean_x >.));
RUN ;
```

Ausgabe:

```
Obs     mean_x

20      0.45901
```

Jetzt enthält die Datei `new` nur noch eine Beobachtung mit der gewünschten Kennzahl.

Erzeugung von zwei oder mehreren Teil- bzw. Untermengen in zwei bzw. mehreren Data-Steps:

```
DATA new_M ; SET old (KEEP= ...) ;
IF gender=1 ;
RUN ;
DATA new_F ; SET old (KEEP= ...) ;
IF gender=2 ;
RUN ;
```

Bei großen Datentabellen dagegen ist es **effizienter** alles in einem einzigen Data-Step abzuwickeln:

```
DATA new_M new_F ; SET old (KEEP= ...) ;
SELECT (gender) ;
WHEN (1) OUTPUT new_M ;
WHEN (2) OUTPUT new_F ;
OTHERWISE ;
END ;
RUN ;
```

3.3 Die Anzahl der Beobachtungen kann auch reduziert werden mit Hilfe der Data-Set-Optionen `OBS=` (Anzahl) und `FIRSTOBS=` (Startwert).

Beispiel:

```
DATA new ; SET old (FIRSTOBS=101 OBS=1000) ;
. . .
RUN ;
```

Ab der Beobachtung mit der lfd. Nummer 101 werden 1000 Beobachtungen in die Datei `new` eingelesen.

Anwendung: Man verwende eine temporäre Datei, die nur noch die erforderlichen Beobachtungen enthält. Dies reduziert bei großen Datentabellen dann erheblich die Anzahl der I/O-Operationen.

3.4 Nutze „temporäre“ Arrays anstatt Variablen zu erzeugen und anschließend wieder zu löschen:

```
DATA index ; INPUT ... ;

ARRAY testvar {4} t1-t4 ;
ARRAY cutoffs {4} c1-c4 (20 40 60 80) ;
DO i=1 TO 4 ;
IF testvar{i}>=cutoffs{i} THEN ... ;
END ;
DROP c1-c4 ;
RUN ;
```

Effektiver ist es dagegen die Cutoff-Points über ein temporäres Array zu definieren:

```
DATA index ; INPUT ... ;

ARRAY testvar {4} t1-t4 ;
ARRAY cutoffs {4} _TEMPORARY_ c1-c4 (20 40 60 80) ;
DO i=1 TO 4 ;
IF testvar{i}>=cutoffs{i} THEN ... ;
END ;
RUN ;
```

3.5 Das LENGTH-Statement reduziert ebenfalls die Beobachtungsgröße und den erforderlichen Speicherbedarf pro Variable. Indirekt werden dadurch auch die Anzahl I/O Operationen in einem Datenverarbeitungsprozeß reduziert. Vorsicht ist aber bei der Anwendung eines RISC-Prozessors angebracht. Hier können u.U. „kurze“ numerische Daten zu einem größeren Aufwand führen.

3.6 Nutze die Möglichkeit übersetzte Programme abzuspeichern und später auszuführen:

Standard-Beispiele:
Verschiedene Datentabellen sollen mit den gleichen SAS-Data-Steps bearbeitet werden:

```
LIBNAME lib 'D:\...\...' ;
DATA neu1 ;                /* Data Set 1 */
SET lib.old1 ;

. . . weitere Rechenschritte

RUN ;
DATA neu2 ;                /* Data Set 2 */
SET lib.old2 ;

. . . weitere Rechenschritte

RUN ;
```

Einfacher ist es das übersetzte Programm abzuspeichern mit den Statements:

```
LIBNAME lib  'D:\...\...' ;
LIBNAME plib 'D:\...\...' ;

DATA neu1 ;
SET lib.old1 ;
. . . ;
RUN pgm=plib.prog1 ;
```

und später dann mit verschiedenen Ein- und Ausgabedateien auszuführen:

```
DATA pgm=plib.prog1 ;
REDIRECT INPUT lib.old1=lib.old1 ;
REDIRECT OUTPUT neu1=neu1 ;
RUN ;

DATA pgm=plib.prog1 ;
REDIRECT INPUT lib.old1=lib.old2 ;
REDIRECT OUTPUT neu1=neu2 ;
RUN ;
```

2.2 Optimierung der System Performance

2.2.1 Optimierung der Speichernutzung

Auch hier sind einige der Punkte, wie sie in Abschnitt 2.1 bereits gezeigt wurden, wichtig:

- Lies nur die Felder, die benötigt werden
- Speichere nur die Variablen, die gebraucht werden
- Verkürze die Datenwerte durch Verwendung von Formaten
- Verwende Character anstelle numerischer Variablen
- Speichere Zahlen als 1-Byte Character-Werte
- Verwende `_NULL_` -Data-Sets

Beispiel:

```
DATA new ; SET old (KEEP=... WHERE=(...) ) END=fin ;
FILE PRINT ;
sum_x + x ;  nn+1 ;
IF fin THEN DO ;
            mean_x=sum_x/nn ;
            PUT 'Mean: ' mean_x 12.2 ;
            END ;
DROP sum_x nn ;
RUN ;
```

Dieser Data-Step wird eigentlich ja nur dazu verwendet, den Mittelwert zu bestimmen und das Ergebnis ins OUTPUT-Fenster auszugeben. Allerdings wird hier eine neue SAS Datentabelle erzeugt, in der lediglich die Variablen der bereits vorhandenen SAS-Datentabelle `old` stehen. Dies aber kann effektiver in einem `NULL`-Data-Step erledigt werden, bei dem keine neue SAS-Datei angelegt wird:

```
DATA _NULL_ ; SET old (KEEP=... WHERE=(...) ) END=fin ;
FILE PRINT ;
sum_x + x ;  nn+1 ;
IF fin THEN DO ;
            mean_x=sum_x/nn ;
            PUT 'Mean: ' mean_x 12.2 ;
            END ;
RUN ;
```

Die Ausgabe ist analog zu dem in Beispiel 3.2 bereits gegebenen Musterprogramm.

2.2.2 Optimierung der CPU-Performance

Erneut sind auch hier wieder einige der Möglichkeiten zu nennen, wie sie oben bereits beschrieben wurden:

- Lies nur die Felder, die benötigt werden,
- Speichere die Daten in eine SAS-Datei,
- Verwende nur die Daten / Variablen, die benötigt werden,
- Verwende Informate für Datentransformationen,
- Bei logische Operatoren: Verwende besser den `IN` als den `OR` Operator,
- `DO`-Schleifen: Verwende die Schleife mit der geringsten Iterationsanzahl am weitesten außen,
- Verwende SAS-Funktionen anstelle „eigener“ Programmschritte, z.B.

  ```
  mean_xyz=MEAN(OF x y z) ;
  ```

  anstatt

  ```
  mean_xyz=(x+y+z)/3 ;
  ```

- Verwende Macros für die immer gleichen Programmschritte,
- Benutze `_NULL_`-Data Sets,
- Nutze, soweit möglich, ein `CLASS`-Statement in Prozeduren (nicht alle Prozeduren erlauben ein `CLASS`-Statement!):

Beispiel:
Standardkennzahlen sollen sortiert nach einer Gruppenvariable bestimmt werden:

```
PROC SORT  DATA=new ; BY group ; RUN ;
PROC MEANS DATA=new ; BY group ; RUN ;
```

Alternative mit Hilfe des `CLASS`-Statements:

```
PROC MEANS DATA=new ; CLASS group ;
RUN ;
```

Vorteil: Das `CLASS`-Statement hat keinen permanenten Effekt auf die Daten. Es benötigt zwar etwas mehr Zeit als die Verwendung von `BY`, bei großen Datenmengen aber deutlich weniger als ein separater Sortierschritt!

3 Sortieren – Die Prozedur SORT und Alternativen

Als erster wichtiger Punkt wäre hier zu erwähnen:

- Plane das Sortieren von Data Sets:

  Beispiel:

```
PROC SORT  DATA=new ; BY group ; RUN ;
PROC MEANS DATA=new ; BY group ;
VAR a b c y1 y4 y10 ;
RUN ;

PROC SORT  DATA=new ; BY sex   ; RUN ;
PROC CHART DATA=new ; BY sex   ;
HBAR x1 ;
RUN ;

PROC SORT  DATA=new ; BY group ; RUN ;
PROC PRINT DATA=new ; BY group ;
VAR y1-y10 ;
RUN ;

PROC SORT  DATA=new ; BY sex agegr ; RUN ;
PROC MEANS DATA=new ; BY sex agegr ;
VAR z1-z5 ;
RUN ;
```

  Effektiver ist es folgende Reihenfolge zu verwenden und Prozeduren mit gleichen BY-Gruppenvariablen zusammenzufassen:

```
PROC SORT  DATA=new ; BY group ; RUN ;
PROC MEANS DATA=new ; BY group ;
VAR a b c y1 y4 y10 ;
RUN ;
PROC PRINT DATA=new ; BY group ;
VAR y1-y10 ;
RUN ;

PROC SORT  DATA=new ; BY sex agegr ; RUN ;
PROC CHART DATA=new ; BY sex   ;
HBAR x1 ;
RUN ;
```

```
PROC MEANS DATA=new ; BY sex agegr ;
VAR z1-z5 ;
RUN ;
```

- Optimierung der CPU-Performance bei der Prozedur SORT:

  Prozedur SORT Standard-Aufruf:

  ```
  PROC SORT DATA=large;
  BY group ;
  RUN ;
  ```

  Oder noch eine Kleinigkeit besser: Verwende die Option NOEQUALS beim Sortieren:

  ```
  PROC SORT DATA=large NOEQUALS ;
  BY group ;
  RUN ;
  ```

  Vorteil: Die Reihenfolge weiterer Variablen bleibt dabei nicht erhalten (Nachteil?).

- System (Prozedur) Optionen:

  Manchmal sind auch Modifikationen von System-Optionen ganz nützlich, z.B.:

  ```
  SORTSIZE=n|nK|nM|nG|MIN|MAX|hex
  ```

  Hierbei spezifiziert n|nK|nM|nG|hex die Größe in Byte, Kilobyte, Megabyte, Gigabyte oder als Hexadezimalzahl die Speichergröße, die für die Prozedur SORT zur Verfügung stehen soll. Der entsprechende Wert sollte kleiner als die physikalisch verfügbare Größe gewählt werden. Eine hier geeignet gewählte Größe kann in jedem Fall helfen die Sortierperformanz verbessern zu helfen.

- Alternativen für große Datenmengen:

  Manchmal kann die Host-Sortierroutine schneller sein. Um sie zu aktivieren, muss die SAS-System-Option

  ```
  OPTIONS SORTPGM=host ;
  ```

  verwendet werden. Sie wird allerdings ignoriert, wenn die Option TAGSORT in der Prozedur SORT verwendet wird. Eine weitere Alternative ist es die Prozedur SYNCSORT® zu lizensieren, die schneller als die Standardsortierroutine abläuft.

- TAGSORT-Option:

  Beispiel zur Verwendung der TAGSORT-Option:

  Eine SAS-Datei soll nach 12 Variablen sortiert werden.

```
PROC SORT TAGSORT DATA=dat.m1 ;
BY v1 v2 v3 v4 v5 v6 v7 v8 v9 v10 v11 v12 ;
RUN ;
```

  Wirkung: Es werden nur die Sortierschlüssel und Beobachtungsnummern (=„tags“) in einem temporären SAS-File abgelegt. Am Ende werden mit Hilfe dieser tags die Datensätze aus der Originaldatei in die sortierte Reihenfolge gebracht.

  Diese Prozedur-Option bietet Vorteile in der Performanz, wenn die totale Länge der Sortierkeys klein ist verglichen mit der Satzlänge. Vor allem der benötigte temporäre Speicherplatz beim Sortieren wird dramatisch reduziert.

- Sortieren mit der Prozedur SQL:

  Auch mit Hilfe der SAS-Prozedur SQL kann man Dateien sortieren.

  Beispiel:

```
PROC SQL ;
CREATE TABLE dat.sqlt1 AS
SELECT * FROM dat.m1
ORDER v1, v2, v3, v4, v5, v6, v7, v8, v9, v10, v11, v12 ;
QUIT ;
```

  Es wird aus der Datei dat.m1 eine neue, nach den Variablen v1 bis v12 sortierte SAS-Datei mit dem Namen dat.sqlt1 erzeugt.

- Sortieren mit Hilfe der Prozedur DATASETS und einem Index:

  Beispiel:

```
PROC DATASETS LIBRARY=dat ;
MODIFY m1 ;
INDEX CREATE idxlist=(v1 v2 v3 v4 v5 v6 v7 v8 v9 v10 v11 v12 ) ;
RUN ;
```

  Dieses Programm erzeugt einen Index (Name: idxlist), der so genutzt werden kann, als ob die Datei vollständig sortiert ist. Indizieren ist schneller als Sortieren! Nachteilig kann es u.U. sein, dass ein großer Indexfile erzeugt wird (von einer Größe bis zu 30% der zu sortierenden Datei). Dies bedeutet, dass die Verbesserung in der Geschwindigkeit wiederum zu Lasten der Speicherkapazität geht.

Literatur

[1] SAS Institute Inc. SAS Procedures Guide, Version 8. Cary, NC: SAS Institute Inc.

[2] SAS Institute Inc. SAS Language Reference: Dictionary, Version 8. Cary, NC: SAS Institute Inc.

[3] SAS Institute Inc. SAS/STAT User's Guide, Version 8. Cary, NC: SAS Institute Inc.