Vergleich verschiedener Möglichkeiten des Table-Lookups für größere Tabellen

Ralf Minkenberg
Input Clinical Research GmbH
Lütticher Str. 281
52074 Aachen
r.minkenberg@input-cro.de

Zusammenfassung

Sehr häufig tritt in verschiedenen Anwendungsgebieten das Problem auf, aus einer oft sehr großen (Master-)Tabelle bestimmte Datensätze anhand von Schlüsselvariablen einer meistens eher kleineren (Lookup-)Tabelle zu extrahieren. In diesem Beitrag werden neben einigen klassischen Lösungsmöglichkeiten, z.B. mittels Sortieren oder Indizierung der Datensätze, wesentlich effizientere, nicht so häufig anzutreffende, Methoden dargestellt, mit denen sowohl Zeit als auch Speicher gespart werden können. Ab Version 9 des SAS-Systems ist mit der sog. Hash-Sortierung ein Verfahren im DATA-Step implementiert, das neue mächtige Möglichkeiten für schnelles und effizientes Lookup bietet.

Schlüsselworte: Sortierung, Index, Mergen, Lookup, Hash-Sort, Format.

1 Einleitung

In vielen Problemstellungen liegt häufig eine (sehr) große (Master-)Tabelle mit allen prinzipiell interessierenden Beobachtungen vor, aus der für konkrete Fragestellungen oft nur eine (wesentlich kleinere) Teilmenge von Beobachtungen ausgewählt werden muss. Dieses Verfahren, oft als Lookup bezeichnet, ist bei großen Datensätzen sehr zeit- und speicherplatzintensiv. Den entsprechenden Code zu optimieren ist daher sehr wichtig, erst recht, wenn in einem Projekt mehrere, ähnliche Probleme auftreten. Die Laufdauer von vielen SAS-Programmen kann durch die Verwendung von entsprechend effizientem Code signifikant verkürzt werden. Häufig sind jedoch die verschiedenen Möglichkeiten, ein Lookup durchzuführen, nicht genügend bekannt und so wird viel unnötige Zeit verschenkt. Im folgenden werden anhand einer numerischen Schlüsselvariablen die Beobachtungen einer sog. Lookup-Tabelle (ca. 40.000 Beobachtungen) aus einer ca. 5 Millionen Beobachtungen umfassenden Master-Tabelle mit allgemeinen Informationen zu den Beobachtungen ausgewählt. In beiden Tabellen kommt ein bestimmter Wert der Schlüsselvariable höchstens einmal vor. Dabei werden folgende sieben Verfahren vorgestellt und verglichen:

1. DATA-Step mit MERGE
2. Anlegen eines Index
3. PROC SQL

4. FORMAT-Prozedur und PUT-Befehl
5. SET-Befehle mit der KEY-Option
6. CALL EXECUTE
7. Hash-Sortierung

Alle vorgestellten Verfahren benötigen nur BASE SAS-Befehle, die Hash-Sortierung ist ab Version 9 implementiert.
Die verschiedenen Methoden werden verglichen nach der benötigten System- und User-CPU-Zeit, nach der tatsächlichen Zeit und des beanspruchten Speichers (alle Informationen sind nach Angabe der Option FULLSTIMER im Log angegeben).

2 Klassische Verfahren

2.1 DATA-Step mit MERGE

Das bekannteste und sicher in den meisten Fällen verwendete Verfahren ist das Mergen der beiden betroffenen Tabellen in einem DATA-Step, nachdem beide Tabellen nach der Schlüsselvariablen (in unserem Falle NR) sortiert wurden. Ein entsprechendes Programm sieht folgendermaßen aus:

```
proc sort data=gross; by nr; run;

proc sort data=klein; by nr; run;

data look;
  merge klein(in=a) gross;
  by nr;
  if a;
run;
```

Mit der IN-Variable werden hier die gewünschten Beobachtungen aus der Lookup-Tabelle KLEIN in der Master-Tabelle GROSS identifiziert und in die Ergebnistabelle LOOK geschrieben. Alle Beobachtungen der zu mergenden Datensätze werden hier sequentiell eingelesen und danach entsprechend gefiltert.

2.2 DATA-Step mit MERGE und Index

Um im vorigen Programm das Sortieren der größeren Master-Tabelle GROSS, gerade bei mehrmaliger Verwendung, zu vermeiden, kann zunächst auf dieser Tabelle ein entsprechender Index gesetzt werden. Dies geschieht z.B. mit Hilfe der Prozedur DATASETS:

```
proc datasets library=work memtype=data nolist;
  modify gross;
    index create nr / unique;
  run;
quit;
```

Beim Mergen der beiden Tabellen muss dann nur noch die kleinere Lookup-Tabelle sortiert werden:

```
proc sort data=klein; by nr; run;

data look;
  merge klein(in=a) gross;
  by nr;
  if a;
run;
```

Da das Anlegen eines Indexes relativ zeitaufwendig ist, sollte dies nur dann verwendet werden, wenn mehrere Operationen mit der Indexvariable im Programm vorgesehen sind. Ersparnis ist hier vor allem beim benötigten Speicherplatz zu erzielen.

2.3 PROC SQL

Als Alternative zum DATA-Step wird häufig auch die Prozedur SQL benutzt, innerhalb der natürlich auch ein Table-Lookup leicht möglich ist:

```
proc sql;
  create table look as select gross.* from klein, gross where
    klein.nr = gross.nr;
quit;
```

Eine Optimierung dieses Verfahrens ist unter Umständen sowohl durch vorheriges Sortieren der beiden Tabellen als auch durch Anlegen eines Indexes auf die Master-Tabelle GROSS möglich. Bei dem hier stattfindenden sog. „inner join“ zwischen zwei Tabellen wird das kartesische Produkt der beiden Tabellen auf diejenigen Beobachtungen eingeschränkt, für die die angegebene WHERE-Bedingung erfüllt ist.

Bei Verwendung von SQL kann das Table-Lookup häufig wesentlich schneller durchgeführt werden, wenn statt des „inner joins“ mit einer entsprechenden Unterabfrage („Subquery“) gearbeitet wird:

```
proc sql;
  create table look as select * from gross where nr in (select nr
    from klein);
quit;
```

Durch die Verwendung einer Unterabfrage, die dann wiederum Teil einer äußeren Abfrage ist, muss kein kartesisches Produkt zwischen den beiden Tabellen gebildet werden, sondern es werden Zeilen einer Tabelle, ausgehend von Werten einer anderen Tabelle, ausgewählt. Auch hier kann evtl. die Verwendung eines Indexes die Ausführung beschleunigen.

3 Weitere effiziente Verfahren

3.1 FORMAT-Prozedur und PUT-Befehl

Neben den im vorigen Kapitel vorgestellten Verfahren zum Table-Lookup gibt es weitere, eher nicht so offensichtliche Möglichkeiten, einen solchen, oft sogar in kürzerer Zeit mit weniger Speicher, durchzuführen (siehe auch [1]). Eine erste solche Methode ist die folgende über das Anlegen eines passenden Formats mittels PROC FORMAT.

Hierbei wird ein Format-Datensatz aus der Lookup-Tabelle erzeugt, also dynamisch ein Format erzeugt, mit dem das Auslesen der benötigten Beobachtungen aus der Master-Tabelle ermöglicht wird:

```
data fmtin;
  set klein(rename = (nr = start));
  retain label 'KLEIN' fmtname 'NR';
run;

proc format cntlin=fmtin;
run;

data look;
  set gross;
  where put(nr, nr.) eq 'KLEIN';
run;
```

Durch den WHERE-Befehl werden nur die tatsächlich gewünschten Beobachtungen eingelesen und die Master-Tabelle muss nicht komplett eingelesen werden.

3.2 SET-Befehle mit der KEY-Option

Die folgende Methode ist eher schwieriger zu verstehen, erweist sich jedoch als äußerst wirkungsvoll, wenn die Master-Tabelle sehr groß und indiziert ist. In einem ersten SET-Befehl wird die kleinere Lookup-Tabelle sequentiell eingelesen. Anschließend wird in einem zweiten SET-Befehl die indizierte Master-Tabelle unter Hinzufügen der KEY-Option eingelesen. Dies geschieht so nur auf Grundlage der Schlüsselvariable, also werden auch nur die Beobachtungen, die aufgrund der ersten Lookup-Tabelle gewünscht sind, eingelesen. In die Ergebnistabelle gelangen Beobachtungen also nur bei Übereinstimmung der Schlüsselvariable in den beiden eingelesenen Tabellen.

```
data look;
  set klein;
  set gross key=nr;
run;
```

Bei dieser Methode ist zu beachten, dass bei mehrfach auftretenden gleichen Werten für die Schlüsselvariable in der Lookup-Tabelle KLEIN das Resultat im Allgemeinen nicht dem Gewünschten entspricht.

3.3 CALL EXECUTE

Ein weiteres äußerst effizientes Verfahren lässt sich mittels des Befehls CALL EXECUTE realisieren. Hier wird über einen DATA _NULL_-Step der nach diesem Step auszuführende Code in der Weise erzeugt, dass die einzelnen Werte der Schlüsselvariable in einem IN-Befehl aufgelistet werden:

```
data _null_;
  set klein end=last;
  if _n_ eq 1 then do;
    call execute('data look;');
    call execute('set gross;');
    call execute('where nr in (');
  end;
  call execute(nr);
  if last then do;
    call execute(');');
    call execute('run;');
  end;
run;
```

Der durch die CALL EXECUTE-Befehle erzeugte SAS-Code wird direkt automatisch im Anschluss an den oben angegebenen _NULL_-DATA-Step ausgeführt. Dieses Verfahren funktioniert nur, wenn nicht zu viele Werte der Schlüsselvariablen ausgewählt werden sollen, da sonst das Programm aufgrund einer zu langen zu erzeugenden Programmzeile mit einer Fehlermeldung abbricht (`ERROR: An internal expression limit of 32767 entries in a list has been reached. This expression cannot be parsed.`). Ist diese Einschränkung in der Anzahl der Schlüsselwerte jedoch erfüllt, hat man hier ein sehr schnelles, effizientes Verfahren.

4 Hash-Sortierung

4.1 Das Prinzip des Hashing

Unter Hashing versteht man allgemein einen Prozess, der einen relativ langen Schlüsselwert (beliebigen Aussehens) durch eine bestimmten mathematischen Algorithmus oder eine mathematische Funktion abbildet auf eine kürzere ganze Zahl aus einer möglichst kleinen Menge ganzer Zahlen.

Im folgenden soll das Prinzip an Hand eines einfachen Beispiels verdeutlicht werden. In einer Tabelle seien 10 Beobachtungen, eindeutig gekennzeichnet durch einen dreistelligen numerischen Schlüsselwert, gegeben. Beispielsweise:

```
data bsp;
  input key @@;
  cards;
185 971 400 260 922 970 543 532 050 067
  ;
run;
```

Idealerweise würde man jetzt eine Hash-Funktion finden wollen, die den gegebenen dreistelligen Schlüsselwert eineindeutig auf eine ganze Zahl von 0 bis 9 abbildet, dies wäre eine perfekte Hash-Funktion. Die Bestimmung einer solchen perfekten hash-Funktion gestaltet sich im allgemeinen sehr aufwendig, ist meistens schwer zu programmieren und muss bei zusätzlichen Beobachtungen, in unserem Beispiel bei nur einer zusätzlichen Beobachtung, neu bestimmt werden. Aus diesen Gründen wird in der Praxis eine nicht perfekte Hash-Funktion verwendet. Eine solche lässt sich leichter und schneller bestimmen, es kann jedoch keine „eins zu eins“-Beziehung zwischen ursprünglichen Schlüsselwert und Hash-Wert garantiert werden. Bei somit auftretenden „Kollisionen“ muss natürlich noch eine Methode zur Auflösung dergleichen angegeben werden. Ein beliebtes, einfaches Beispiel einer Hash-Funktion ist das Teilen der gegebenen Schlüsselwerte durch eine Primzahl.

Was muss nun beachtet werden, um Hashing effektiv nutzen zu können?

1. Welcher Anteil der möglichen Spannweite an ganzen Zahlen soll tatsächlich nach dem Hashing mit Werte belegt sein? Je weniger der möglichen Zahlen nach dem Hashing belegt sind, je unwahrscheinlicher ist das Auftreten von Kollisionen, aber je mehr Speicher wird benötigt. Als für viele Zwecke praktikabel hat sich ein nachher belegter Anteil von ca. 80% erwiesen.
2. Wenn der gewünschte Beleganteil nach dem Hashing vorgegeben wird, muss nun die optimale Spannweite für das Bild der Hash-Funktion bestimmt werden. Zum Glück gibt es hierfür etablierte Algorithmen.

Geht man von dem vorgeschlagenen Anteil an belegten Werten von 80% aus, ergibt sich in unserem kleinen Beispiel als sog. Hash-Größe 13, d.h. die vorgegebenen Schlüssel werden durch 13 geteilt und der verbleibende Rest wird als Ergebnis genommen (Modulo-Rechnen). Es ergibt sich folgende Zuordnung:

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

Es ist zu sehen, dass sich hier zwei Kollisionen ergeben haben. Sowohl der Schlüsselwert 400 als auch 543 werden auf die 11 abgebildet, und sowohl 922 als auch 532 werden auf die 13 abgebildet. Das Auflösen solcher Kollisionen kann aufgrund verschiedener Algorithmen vorgenommen werden. Dabei muss für den zweiten und alle weiteren auf ein und denselben Wert abgebildeten Schlüsselwerte mit möglichst wenig Aufwand eine „freie“ Zahl gefunden werden. (siehe [2])

Mit Hilfe des Hashings ist es möglich, Suchvorgänge und Tabellen-Lookups so effizient wie nur denkbar zu gestalten, denn durch die Abbildung kann jedes interessierende Element, jeder Schlüsselwert direkt durch seine Position in der Tabelle angesprochen werden. Im nächsten Unterabschnitt wird dieses Verfahren nun in SAS verwendet.

4.2 Das Prinzip des Hashing

Um in SAS ab Version 9 ein Tabellen-Lookup unter Verwendung von Hash-Funktionen durchzuführen, muss innerhalb des DATA-Steps objekt-orientiert programmiert werden. Für die in den Kapiteln 2 und 3 verwendeten Beispieltabellen sieht dies folgendermaßen aus:

```
data look(drop = rc);
  length nr $12 gross $8;
  declare Hash ha();
  rc = ha.DefineKey('nr');
  rc = ha.DefineDone();
  do until (last1);
    set klein end=last1;
    rc = ha.add();
  end;
  do until (last2);
    set gross end=last2;
    rc = ha.find();
    if rc eq 0 then output;
  end;
  stop;
run;
```

Nachdem über den LENGTH-Befehl die Längen der notwendigen Datenelemente für die folgenden Methodenaufrufe festgelegt werden, wird mit DECLARE die Hash-Tabelle HA definiert. In der DefineKey-Methode wird die Schlüsselvariable definiert und mit DefineDone wird die Initialisierung des Hash-Objekts abgeschlossen. Mit der Add-Methode wird nun ein Schlüsselwert aus der KLEIN-Tabelle in die Hash-Tabelle geladen, mit der FIND-Methode wird nun nach passenden Schlüsselwerten in der GROSS-Tabelle gesucht und bei Matching ausgegeben.

Die vielleicht noch ein wenig umständliche Verfahrensweise in obigem Beispiel lässt sich noch wie folgt vereinfachen:

```
data look;
  set klein point=_n_;
  declare Hash ha(dataset:'work.klein', hashexp:10);
  ha.DefineKey('nr');
  ha.DefineDone();
  do until (last);
    set gross end=last;
    if ha.find() eq 0 then output;
  end;
  stop;
run;
```

Die Eigenschaften der benötigten Daten für die Methoden werden hier durch Einlesen einer Zeile aus der KLEIN-Tabelle festgelegt. Die declare-Anweisung wird erweitert, indem sowohl der Lookup-Datensatz als auch die Anzahl maximaler Hash-Objekte (hier 2**10=1024) als Optionen mitgegeben werden.

Mit den in SAS 9 neu implementierten Hash-Funktionen ist es möglich, Tabellen-Lookups sehr effizient durchzuführen und sowohl unter Zeit- als auch unter Speichergesichtspunkten sehr effektiv zu arbeiten. Die weiteren Möglichkeiten der Hash-Funktionen und Details der oben vorgestellten Syntax würden den Rahmen sprengen, es muss auf entsprechende Literatur verwiesen werden. (siehe [3])

5 Vergleich der Methoden

Im folgenden ist der Zeit- und Speicherverbrauch der verschiedenen bisher vorgestellten Verfahren dargestellt. Für alle Beispielprogramme wurde der Datensatz KLEIN mit ca. 40.000 Beobachtungen und nur der Schlüsselvariablen (numerisch), sowie der Datensatz GROSS mit ca. 5.000.000 Beobachtungen, der Schlüsselvariablen und drei weiteren Variablen verwendet. Es wurde (durch Angabe der Option FULLSTIMER) unterschieden nach System-CPU-Zeit, User-CPU-Zeit, tatsächliche Zeit und Speicher.
Die Verfahren sind in den Graphiken folgendermaßen nummeriert:

1	DATA-Step mit Merge (siehe 2.1)
2	DATA-Step mit Merge und Index (siehe 2.2)
4	PROC SQL inner join (siehe 2.3)
6	PROC SQL subquery (siehe 2.3)
8	FORMAT-Prozedur udn PUT-Befehl (siehe 3.1)
9	SET-Befehle mit KEY-Option (siehe 3.2)
11	Hashing (siehe 4.2)

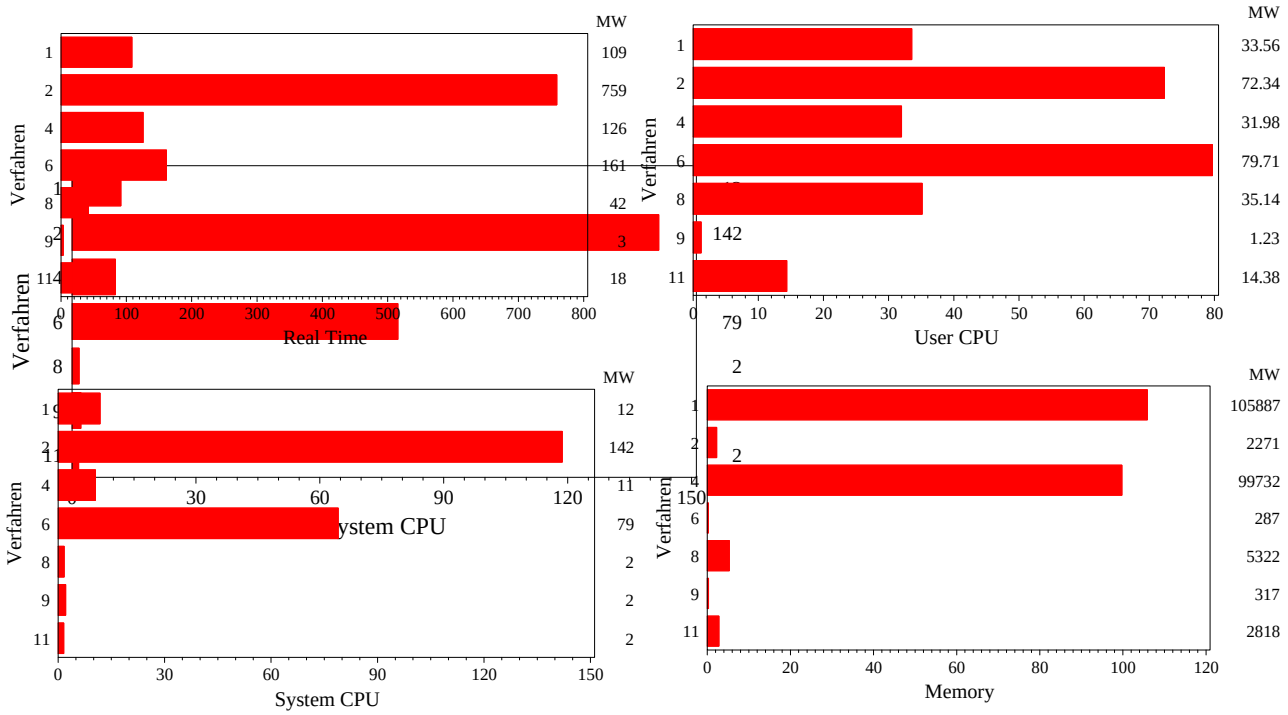

Abbildung 1: Zeit- und Systemverbrauch verschiedener Lookup-Verfahren

Am effizientesten sind das Hash-Verfahren (das ohne Index auskommt) und das Verfahren mittels SET- und KEY-Befehlen bei Vorliegen eines Indexes. Beide sind vor allem den bekannten Verfahren erheblich überlegen.

Literatur

[1] Croonen, N., Thauwissen, H. (2002): *Table Lookup: Techniques Beyond the Obvious*, Paper 11-27 SUGI 27

[2] D[illegible], G. (2002): *Hashing Rehashed*, Paper 12-27 SUGI 27

[3] D[illegible] *Hash Components Objects: Dynamic Data St[illegible]Look-Up*, Paper 238-29 SUGI 29