The Stata Journal (2008)
8, Number 4, pp. 540–553

A shortcut through long loops: An illustration of two alternatives to looping over observations

Ward Vanlaar
Traffic Injury Research Foundation
Ottawa, Canada
wardv@trafficinjuryresearch.com

Abstract. It is well known that looping over observations can be slow and should be avoided. The objective of this article is to discuss two alternative solutions to looping over observations that can be used to overcome a particular data-management problem of merging datasets in which unique key identifiers changed over time. The first alternative, `mapch`, which is introduced in this article, uses a combination of appending, indexing, and merging to solve the problem, while the second alternative uses repeated merging. Both solutions are much quicker than looping over observations. However, depending on the nature of the problem, one solution may work better than the other. It is argued that the use of such dataset-type manipulations may be suitable to overcome other data-management problems. More generally speaking, the issue that is addressed—searching for an alternative to looping over observations—may be common and illustrates the importance of balancing the costs of developing an efficient solution with the benefits accruing from that solution.

Keywords: dm0041, mapch, appending, data management, indexing, looping, merging

1 Introduction

It is well known that looping over observations can be slow and should be avoided. Cox, for example, lists this as one of his suggestions regarding Stata programming style on speed and efficiency; he comments that "fortunately, [looping over observations] can usually be avoided" (2005, 565). This article elaborates on this suggestion and illustrates how the issue regarding time constraints can be overcome by relying on such alternatives as appending, indexing, and merging.

First, the problem—essentially, the issue of unique key identifiers that changed over time—will be described. Following this description, two practical solutions will be discussed as alternatives to looping over observations. These solutions will also be compared and contrasted. Finally, some general conclusions will be drawn from the central ideas that were applied, and some thought will be given to balancing the costs associated with the development of an efficient solution with the benefits accruing from that solution.

2 The problem

The illustration in this article is based on the problem of merging datasets from a particular jurisdiction in which unique key identifiers, more precisely, driver license numbers, changed over time. A substantial portion of the jurisdiction's population of several hundred thousand drivers changed their driver license numbers—many of them more than once—because the driver license number is an alphanumeric combination based on the driver's name. Furthermore, the jurisdiction's data warehouse is organized such that certain datasets use old driver license numbers, for example, the number in use at the time of a crash, while other datasets are constantly updated and use only the most recent driver license number.

As a result, the former datasets may contain multiple records per driver, for example, if a driver was involved in more than one crash. Each such record may be listed under a different driver license number; for example, if a driver was involved in two crashes and the second crash happened after the driver license number changed, each record pertaining to this driver would be listed under a different driver license number. The datasets using only the most recent number contain records of those same drivers, but they are listed using only one driver license number, more precisely, the driver license number in use at the time of data extraction. Using nonupdated driver license numbers as key identifiers in some datasets and updated driver license numbers as key identifiers in other datasets may become problematic when merging datasets of both kinds.

For the sake of clarity in this article, I will refer to the old, nonupdated driver license numbers as *the old numbers*; one driver can be listed under several old numbers. I will refer to the updated, most recent number as *the updated number* or *terminal ID*; one driver can be listed under only one such updated number.

The involved jurisdiction provided a cross-reference dataset containing three variables: a variable with the old driver license number, a variable with the updated driver license number, and a variable indicating when the old number changed into the new number. As such, the cross-reference dataset from the involved jurisdiction contains approximately 86,000 lines. Each line represents a change pertaining to a certain driver, and there may be several lines per driver; more precisely, if a driver changed his or her driver license number more than once, there would be more than one line for that driver. Example 1 illustrates the format of this cross-reference dataset.

▷ Example 1: Illustration of the format of the cross-reference dataset

```
. use testfileSJ
. sort date
```

(*Continued on next page*)

```
. list old updated date
```

	old	updated	date
1.	A	B	16001
2.	Q	P	16004
3.	E	F	16007
4.	G	H	16008
5.	X1	X2	16016
6.	X2	X3	16017
7.	P	O	16100
8.	O	N	16999
9.	B	C	17000
10.	C	D	17200

◁

Two challenges arise with such a cross-reference dataset if it is to be used to overcome the problem of dataset merging described previously. Because lines of information pertaining to the same person may be spread throughout the entire dataset (e.g., in example 1, lines 1, 9, and 10 all pertain to the same driver), the first challenge is to find a way to sort through this wealth of information and identify each driver's changes, knowing that there may be several changes per driver. This can be particularly challenging with large datasets (e.g., say the first change is on line 1, but the second is not until line 51,000). The process of sorting and finding each driver's changes can be called mapping the chains of events, with an event in this context being a change of the driver license number.

Once the chains have been mapped for each driver, the second challenge is to create a new variable that contains the end value of each chain, or the terminal ID (the updated driver license number).

How to overcome both challenges is explained in more detail in the next section, which illustrates two solutions that can be used as alternatives to looping over observations.

3 The solution

3.1 Conceptual approach

To capture all records per driver when merging datasets containing the updated numbers with datasets containing the old numbers, the changes in driver license numbers have to be mapped; i.e., each driver's changes have to be identified to produce chains (e.g., if "A" changes into "B" and later on "B" changes into "C", then this produces the chain "A B C"). And somehow, each old number pertaining to the same driver has to be linked to the terminal ID (e.g., "A" has to be linked to "C", and "B" also has to be linked to "C"). Such a cross-reference dataset can be merged with datasets containing the old numbers by using the old number as the key identifier. In a second step, the

resulting dataset can then be merged to datasets containing the updated numbers by using the updated number as key identifier (the updated number is contained in each driver record because of the first step).

From a conceptual point of view, this is an easily understood solution for a not too complicated problem. However, the hardest part is translating this conceptual solution into a practical one. By using a combination of several looping procedures on a dataset with 86,000 changes in driver license numbers, some with chains of length of five steps (i.e., some people in this jurisdiction changed their driver license number as many as five times), it took about 12 hours to compare the first 8,000 values from the variable `updated` with the values from the variable `old` in the first loop. This performance review was run on an Intel Pentium M, 1.73 GHz processor with 1.5 GB of RAM. Obviously, this was not a feasible approach. Two alternative solutions were developed: the first one, `mapch`, which I developed, uses a combination of appending, indexing, and merging; the second one—suggested as an alternative to `mapch` by a reviewer—uses repeated merging.

Both approaches are discussed in more detail below and are compared with one another. The time gained by using either alternative rather than looping over observations is astounding: the dataset, containing 86,000 changes, can be mapped in a couple of seconds rather than dozens of hours.

3.2 mapch: appending, indexing, and merging as an alternative to looping over observations

Inside the program mapch

While mapping chains was not at all feasible by looping over observations, fortunately, the mapping can be done by using a combination of appending, indexing, and merging. This approach is formalized in `mapch`. This section explains how `mapch` functions by illustrating how the data patterns that emerge when using a combination of appending, indexing, and merging allow for an efficient means of mapping the chains.

In the first step, the variable `link` is created as a copy of the variable `updated`. This dataset is saved and, using the original dataset, the variable `link` is created again, but this time it is created as a copy of the variable `old`. Both resulting datasets are then appended. Example 2 displays the result of these data manipulations after sorting on `link` and `date`.

(*Continued on next page*)

▷ Example 2: Resulting dataset after appending

```
. use testfileSJ
. sort date
. generate link = updated
. save testfile1, replace
(note: file testfile1.dta not found)
file testfile1.dta saved
. use testfileSJ, clear
. generate link = old
. append using testfile1
. sort link date
. by link: generate test1=_N
. by link: generate test2=_n
. list old updated date link
```

	old	updated	date	link
1.	A	B	16001	A
2.	A	B	16001	B
3.	B	C	17000	B
4.	B	C	17000	C
5.	C	D	17200	C
6.	C	D	17200	D
7.	E	F	16007	E
8.	E	F	16007	F
9.	G	H	16008	G
10.	G	H	16008	H
11.	O	N	16999	N
12.	P	O	16100	O
13.	O	N	16999	O
14.	Q	P	16004	P
15.	P	O	16100	P
16.	Q	P	16004	Q
17.	X1	X2	16016	X1
18.	X1	X2	16016	X2
19.	X2	X3	16017	X2
20.	X2	X3	16017	X3

◁

Despite the noise created by replicating lines of information due to appending (e.g., lines 1 and 2 or lines 7 and 8 in example 2), a useful pattern emerges in the output of `link`. As shown in example 2, `link` contains the value "B" in lines 2 and 3. More generally speaking, each link in a chain can be identified by using this data pattern, which allows us to distinguish between chains of only one change (such as "E" into "F" in lines 7 and 8 of `link`) and chains of at least two changes (such as "A" into "B" into "C"; this time `link` contains "B" in line 2 and in line 3).

The next step consists of cleaning up the noise by dropping each line that is either a replica of another line or a line pertaining to a one-step chain. This can be done by sorting on `link` and then generating a count of lines per value of `link` by using `_N`. If the resulting count is different from two, the observation should be deleted. In example 2, lines 1, 6, 7, 8, 9, 10, 11, 16, 17, and 20 will be deleted. The result of this step can be seen in example 3.

▷ Example 3: Resulting dataset after cleaning up the noise

```
. drop if test1!=2
(10 observations deleted)
. list old updated date link
```

	old	updated	date	link
1.	A	B	16001	B
2.	B	C	17000	B
3.	B	C	17000	C
4.	C	D	17200	C
5.	P	O	16100	O
6.	O	N	16999	O
7.	Q	P	16004	P
8.	P	O	16100	P
9.	X1	X2	16016	X2
10.	X2	X3	16017	X2

◁

Now that we have identified chains with at least two steps, we can create yet another variable, `recent`, that contains the most recent update, using indexing and after sorting on `date`. It is also possible to achieve this if dates are not available. The package `mapch` can be run with or without dates, although I do not illustrate the latter in this article.

After sorting on `date`, the variable `recent` can be generated such that it contains the most recent value of `updated`. For example, for `link` equal to "B", the variable `recent` would contain "C". The result is displayed in example 4.

▷ Example 4: Resulting dataset after generating recent

```
. generate str2 recent=""
(10 missing values generated)
. by link: replace recent=updated[_N]
(10 real changes made)
. sort old date
```

```
. list old updated date link recent
```

	old	updated	date	link	recent
1.	A	B	16001	B	C
2.	B	C	17000	B	C
3.	B	C	17000	C	D
4.	C	D	17200	C	D
5.	O	N	16999	O	N
6.	P	O	16100	O	N
7.	P	O	16100	P	O
8.	Q	P	16004	P	O
9.	X1	X2	16016	X2	X3
10.	X2	X3	16017	X2	X3

◁

At this point, mapping two-step chains is complete. For example, the chain "X1 X2 X3" has been identified as such a two-step chain, and its values for `recent` are "X3", i.e., the end value (terminal ID) of that chain, in lines 9 and 10 in example 4. Three-step and longer chains, on the other hand, have not yet been mapped completely, for example, the three-step chains "A B C D" and "Q P O N". To replace their values of `recent` with the end value of the chain, the process of appending and indexing has to be repeated. However, before it can be repeated, the original dataset has to be restored and extended with the variable `recent`.

To restore the original dataset, more noise has to be cleaned up. For example, lines 3 and 6 in example 4 are replicas of lines 2 and 7, so we need a solution to delete those replicated lines. This can be done by exploiting the data pattern that emerges when sorting on the variable `old`. Whenever the count of lines per value of the variable `old` is greater than one, one of the lines should be dropped. Example 5 shows the result of this data manipulation. Each line displayed in example 4 is also displayed in example 5, except for lines 3 and 6.

▷ Example 5: Resulting dataset after cleaning up more noise

```
. by old: generate test3=_N

. by old: drop if updated!=recent & test3>1
(2 observations deleted)

. save testfile2, replace
(note: file testfile2.dta not found)
file testfile2.dta saved
```

```
. list old updated date link recent
```

	old	updated	date	link	recent
1.	A	B	16001	B	C
2.	B	C	17000	B	C
3.	C	D	17200	C	D
4.	O	N	16999	O	N
5.	P	O	16100	P	O
6.	Q	P	16004	P	O
7.	X1	X2	16016	X2	X3
8.	X2	X3	16017	X2	X3

◁

The dataset in example 5 can now be merged with the original dataset, using `old` as the key identifier. As shown in example 6, one- and two-step chains are completely mapped, but three-step chains are only partly mapped at this point. The process of appending, indexing, and merging can now be repeated to complete the mapping for three-step chains. The variable `link` has to be created by copying the result of `recent` (rather than `updated`) in one dataset and copying `old` in the other dataset before appending both datasets.

▷ **Example 6: Resulting dataset after the first iteration of appending, indexing, and merging has been completed**

```
. use testfileSJ, clear
. sort old
. merge old using testfile2
. replace recent=updated if recent==""
(2 real changes made)
. drop link test1 test2 test3 _merge
. erase testfile1.dta
. erase testfile2.dta
. list old updated date recent
```

	old	updated	date	recent
1.	A	B	16001	C
2.	B	C	17000	C
3.	C	D	17200	D
4.	E	F	16007	F
5.	G	H	16008	H
6.	O	N	16999	N
7.	P	O	16100	O
8.	Q	P	16004	O
9.	X1	X2	16016	X3
10.	X2	X3	16017	X3

◁

Repeat this process as many times as the length of the longest chain in the dataset minus one to map all chains. Actually, because it may not be known in advance what the length of the longest chain will be, the process has to be repeated one more time after all chains have been mapped to allow for a comparison of the number of changes in `recent` with the number of changes in the previous run. When both counts are equal, the process of mapping is complete. Thus the number of times the process has to be repeated is really equal to the length of the longest chain in the dataset. The command `mapch` automatically stops when that point has been reached, and then it creates a dataset, `mapping`, that contains the mapped chains and summarizes how many chains of length n the dataset contains, with $1 \leq n \leq N$.

The output after running `mapch` on the example dataset is contained in example 7, and the resulting dataset, `mapping`, is displayed in example 8. The variable `NoOfEvents` refers to the number of changes per chain and is used to summarize how many chains of length n are contained in the dataset.

▷ Example 7: Output of mapch

```
. use testfileSJ, clear
. mapch old updated date
********************
* Mapping complete *
********************
Frequency of NoOfEvents:
```

NoOfEvents	Freq.	Percent	Cum.
1	2	20.00	20.00
2	2	20.00	40.00
3	6	60.00	100.00
Total	10	100.00	

```
The number of 1-step chains is equal to 2/1
The number of 2-step chains is equal to 2/2
The number of 3-step chains is equal to 6/3
```

◁

▷ Example 8: Resulting dataset, mapping, after mapping chains has been completed

```
. list
```

	old	updated	date	recent	NoOfEv~s
1.	A	B	16001	D	3
2.	B	C	17000	D	3
3.	C	D	17200	D	3
4.	E	F	16007	F	1
5.	G	H	16008	H	1
6.	Q	P	16004	N	3
7.	P	O	16100	N	3
8.	O	N	16999	N	3
9.	X1	X2	16016	X3	2
10.	X2	X3	16017	X3	2

◁

3.3 Repeated merging as a second alternative to looping over observations

An alternative to `mapch` consists of repeatedly match-merging the variable `old` to the variable `updated`, until nothing further changes. This procedure works as follows: In the first step, a clone of `updated` is added to the original file (see example 9a). Then, a clone of `old`, entitled `clone`, and a clone of `updated`, entitled `terminal`, are created and stored in a temporary file (see example 9b).

▷ Example 9a: Original dataset, including a clone of updated

```
. use testfileSJ, clear
. keep in 1/5
(5 observations deleted)
. generate clone=updated
. list old updated clone
```

	old	updated	clone
1.	A	B	B
2.	B	C	C
3.	C	D	D
4.	E	F	F
5.	G	H	H

◁

(*Continued on next page*)

▷ Example 9b: Temporary dataset

```
. use testfileSJ, clear
. keep in 1/5
(5 observations deleted)
. rename old clone
. rename updated terminal
. list clone terminal
```

	clone	terminal
1.	A	B
2.	B	C
3.	C	D
4.	E	F
5.	G	H

◁

The temporary file is then merged with the original file, using the clone of `old` as the key identifier in the temporary file and the clone of the variable `updated` in the original file. The resulting dataset (see example 10) contains the original variables, `old` and `updated`; the unique key identifier, `clone`; and the merged values of `terminal`, coming from the temporary file. These values correspond to the values that are one step further in the chain and can be used to replace the values of `updated` in the original file (see example 11). The process can then be repeated until nothing further changes, thus allowing replacement of the variable `updated` with the terminal IDs, thereby mapping the chains.

▷ Example 10: Resulting dataset after merging the original dataset (figure 9a) with the temporary dataset (figure 9b), using clone as the key identifier

```
. use testfileSJ, clear
. keep in 1/5
(5 observations deleted)
. generate clone=updated
. generate str1 terminal = "C" in 1
(4 missing values generated)
. replace terminal = "D" in 2
(1 real change made)
. list old updated clone terminal
```

	old	updated	clone	terminal
1.	A	B	B	C
2.	B	C	C	D
3.	C	D	D	
4.	E	F	F	
5.	G	H	H	

◁

▷ Example 11: Resulting dataset after replacing the values of updated with the corresponding values of terminal after the first iteration

```
. use testfileSJ, clear
. keep in 1/5
(5 observations deleted)
. replace updated = "C" in 1
(1 real change made)
. replace updated = "D" in 2
(1 real change made)
. list old updated
```

	old	updated
1.	A	C
2.	B	D
3.	C	D
4.	E	F
5.	G	H

◁

3.4 mapch versus repeated merging

Both approaches were tested on the previously mentioned dataset, containing approximately 86,000 changes, and were found to perform equally well: both solutions mapped all chains in a matter of seconds. One advantage of the repeated merging solution is that it does not start by doubling the dataset, unlike `mapch`, which uses appending. While this did not affect the performance of `mapch` with 86,000 changes, it may do so with datasets that contain many more changes.

While doubling the dataset may be less efficient with very large datasets, `mapch` does have the advantage of automatically leading to a verification of the number of changes per case per unit of time. If time of change is used, the `mapch` algorithm is not stable when more than one change for the same case takes place at the same time, and this will be noticed, even if you are not intentionally looking for such an anomaly. For example, in the dataset of changes in driver license numbers that was originally provided, it was found—because of the doubling—that several hundreds of drivers allegedly had changed their driver license number more than once on the same day. It is very unlikely that someone would change his or her driver license number twice on the same day, but these anomalies would probably not have been identified if the repeated merging approach was applied to this particular problem instead of the doubling approach of `mapch`.

Depending on the context that you are working in, checking for the number of changes per case per unit of time may or may not be crucial. For this reason, `mapch` can be run with or without an indication of the time when the changes took place; with an indication of time, `mapch` automatically checks whether this condition is violated and issues an error message, if necessary.

A combination of both approaches (`mapch` and repeated merging)—allowing the user to avoid doubling the dataset at the beginning of every loop yet still ensuring that no two changes for the same case take place at the same time—may be beneficial when dealing with extremely large datasets in a context where two such changes would be indicative of an anomaly with the data.

4 The mapch command

4.1 Syntax

`mapch` *begin end* [*time*] [*if*] [*in*]

4.2 Description

`mapch` maps chains of events. A *chain* consists of at least one event; an *event* in this context is a change of the information contained in the variable *begin* into the information contained in the variable *end*. Optionally, the time at which the event took place can be stored in the variable *time* and used to map the chains chronologically. It is assumed that both *begin* and *end* contain unique information, i.e., each value in both variables can appear only once. It is also assumed that events in a chain cannot occur at the same time and that chains are not circular, i.e., the begin value of a chain must not be the same as the end value of that chain.

`mapch` creates a dataset called `mapping` that contains maps of each chain and two or three additional variables: `recent`, whose value is equal to the end value of the chain for each step in that chain; `date` (only in case real time is not available), a fictitious time when the event took place, allowing the user to sort the information; and `NoOfEvents`, the number of events per chain. `mapch` also tabulates the frequency of n-step chains, with $1 \leq n \leq N$ (N = total number of events in your dataset).

5 Conclusion

It is well known that looping over observations is slow and should be avoided. This article illustrated two alternative solutions to looping over observations in the specific context of a—perhaps somewhat peculiar—data-management problem related to unique key identifiers that had changed over time. While the alternative solutions may have some advantages and disadvantages compared to one another, they both work really well in this particular context and solve the problem much more rapidly compared to looping over observations. The central ideas in this article—creating a "link" variable by appending and then using the resulting data patterns in combination with indexing and merging on the one hand, or repeated merging on the other hand—convincingly show that some dataset-type manipulations are worth considering as part of a solution, given their efficient performance. Using dataset-type manipulations may be useful to overcome other data-management problems as well.

More generally speaking, the need for a viable, efficient alternative to looping over observations is probably quite common. As such, this article bears on the importance of balancing the costs of developing an efficient solution with the benefits accruing from that solution. It could be argued that applying the central idea of `mapch` or the repeated merging approach may be regarded as a detour to solving the problem, while, conceptually, looping over observations is probably more straightforward. However, it was illustrated that investing some time into looking for an efficient albeit less straightforward solution was worth the effort. Eventually, it all comes down to cost/benefit ratios, which may turn out surprisingly favorably, given the vast difference in performance of different solutions, as illustrated in this article.

6 Acknowledgments

I am grateful for the helpful feedback that was provided by Kerry Kammire, a technical services representative at StataCorp, when developing `mapch`. I also extend my gratitude to a reviewer who provided useful comments to improve this article and who suggested another alternative solution to the problem discussed in this article.

7 Reference

Cox, N. J. 2005. Suggestions on Stata programming style. *Stata Journal* 5: 560–567.

About the author

Ward Vanlaar is a research scientist with the Traffic Injury Research Foundation. He teaches quantitative methods in criminology as a part-time professor at the University of Ottawa. Before working at the Traffic Injury Research Foundation, he worked for the Behaviour and Policy Department of the Belgian Road Safety Institute, where he served as Head of Research from 2001 to 2005. His main fields of interest are traffic enforcement issues, the effects of alcohol and drugs on driving, safety performance indicators, risk perception, multilevel/mixed modeling, complex sampling designs, multidimensional scaling, and data management.