Title

stata.com

merge — Merge datasets

Syntax

One-to-one merge on specified key variables

```
merge 1:1 varlist using filename [ , options ]
```

Many-to-one merge on specified key variables

```
merge m:1 varlist using filename [ , options ]
```

One-to-many merge on specified key variables

```
merge 1:m varlist using filename [ , options ]
```

Many-to-many merge on specified key variables

```
merge m:m varlist using filename [ , options ]
```

One-to-one merge by observation

```
merge 1:1 _n using filename [ , options ]
```

options	Description
Options	
`keepusing(`*varlist*`)`	variables to keep from using data; default is all
`generate(`*newvar*`)`	name of new variable to mark merge results; default is `_merge`
`nogenerate`	do not create `_merge` variable
`nolabel`	do not copy value-label definitions from using
`nonotes`	do not copy notes from using
`update`	update missing values of same-named variables in master with values from using
`replace`	replace all values of same-named variables in master with nonmissing values from using (requires `update`)
`noreport`	do not display match result summary table
`force`	allow string/numeric variable type mismatch without error
Results	
`assert(`*results*`)`	specify required match results
`keep(`*results*`)`	specify which match results to keep
`sorted`	do not sort; dataset already sorted

`sorted` does not appear in the dialog box.

Menu

Data > Combine datasets > Merge two datasets

Description

`merge` joins corresponding observations from the dataset currently in memory (called the master dataset) with those from *filename*`.dta` (called the using dataset), matching on one or more key variables. `merge` can perform match merges (one-to-one, one-to-many, many-to-one, and many-to-many), which are often called *joins* by database people. `merge` can also perform sequential merges, which have no equivalent in the relational database world.

`merge` is for adding new variables from a second dataset to existing observations. You use `merge`, for instance, when combining hospital patient and discharge datasets. If you wish to add new observations to existing variables, then see [D] **append**. You use `append`, for instance, when adding current discharges to past discharges.

By default, `merge` creates a new variable, `_merge`, containing numeric codes concerning the source and the contents of each observation in the merged dataset. These codes are explained below in the match results table.

Key variables cannot be `strL`s.

If *filename* is specified without an extension, then `.dta` is assumed.

Options

Options

`keepusing(`*varlist*`)` specifies the variables from the using dataset that are kept in the merged dataset. By default, all variables are kept. For example, if your using dataset contains 2,000 demographic characteristics but you want only `sex` and `age`, then type `merge ..., keepusing(sex age) ...`.

`generate(`*newvar*`)` specifies that the variable containing match results information should be named *newvar* rather than `_merge`.

`nogenerate` specifies that `_merge` not be created. This would be useful if you also specified `keep(match)`, because `keep(match)` ensures that all values of `_merge` would be 3.

`nolabel` specifies that value-label definitions from the using file be ignored. This option should be rare, because definitions from the master are already used.

`nonotes` specifies that notes in the using dataset not be added to the merged dataset; see [D] **notes**.

`update` and `replace` both perform an update merge rather than a standard merge. In a standard merge, the data in the master are the authority and inviolable. For example, if the master and using datasets both contain a variable `age`, then matched observations will contain values from the master dataset, while unmatched observations will contain values from their respective datasets.

If `update` is specified, then matched observations will update missing values from the master dataset with values from the using dataset. Nonmissing values in the master dataset will be unchanged.

If `replace` is specified, then matched observations will contain values from the using dataset, unless the value in the using dataset is missing.

Specifying either `update` or `replace` affects the meanings of the match codes. See *Treatment of overlapping variables* for details.

noreport specifies that merge not present its summary table of match results.

force allows string/numeric variable type mismatches, resulting in missing values from the using dataset. If omitted, merge issues an error; if specified, merge issues a warning.

Results

assert(*results*) specifies the required match results. The possible *results* are

Numeric code	Equivalent word (*results*)	Description
1	master	observation appeared in master only
2	using	observation appeared in using only
3	match	observation appeared in both
4	match_update	observation appeared in both, missing values updated
5	match_conflict	observation appeared in both, conflicting nonmissing values

Codes 4 and 5 can arise only if the update option is specified. If codes of both 4 and 5 could pertain to an observation, then 5 is used.

Numeric codes and words are equivalent when used in the assert() or keep() options.

The following synonyms are allowed: masters for master, usings for using, matches and matched for match, match_updates for match_update, and match_conflicts for match_conflict.

Using assert(match master) specifies that the merged file is required to include only matched master or using observations and unmatched master observations, and may not include unmatched using observations. Specifying assert() results in merge issuing an error if there are match results among those observations you allowed.

The order of the words or codes is not important, so all the following assert() specifications would be the same:

```
assert(match master)
assert(master matches)
assert(1 3)
```

When the match results contain codes other than those allowed, return code 9 is returned, and the merged dataset with the unanticipated results is left in memory to allow you to investigate.

keep(*results*) specifies which observations are to be kept from the merged dataset. Using keep(match master) specifies keeping only matched observations and unmatched master observations after merging.

keep() differs from assert() because it selects observations from the merged dataset rather than enforcing requirements. keep() is used to pare the merged dataset to a given set of observations when you do not care if there are other observations in the merged dataset. assert() is used to verify that only a given set of observations is in the merged dataset.

You can specify both assert() and keep(). If you require matched observations and unmatched master observations but you want only the matched observations, then you could specify assert(match master) keep(match).

assert() and keep() are convenience options whose functionality can be duplicated using _merge directly.

```
. merge ..., assert(match master) keep(match)
```

is identical to

```
. merge ...
. assert _merge==1 | _merge==3
. keep if _merge==3
```

The following option is available with merge but is not shown in the dialog box:

sorted specifies that the master and using datasets are already sorted by *varlist*. If the datasets are already sorted, then merge runs a little more quickly; the difference is hardly detectable, so this option is of interest only where speed is of the utmost importance.

Remarks and examples

stata.com

Remarks are presented under the following headings:

Overview

merge 1:1 *varlist* ... specifies a one-to-one match merge. *varlist* specifies variables common to both datasets that together uniquely identify single observations in both datasets. For instance, suppose you have a dataset of customer information, called customer.dta, and have a second dataset of other information about roughly the same customers, called other.dta. Suppose further that both datasets identify individuals by using the pid variable, and there is only one observation per individual in each dataset. You would merge the two datasets by typing

```
. use customer
. merge 1:1 pid using other
```

Reversing the roles of the two files would be fine. Choosing which dataset is the master and which is the using matters only if there are overlapping variable names. 1:1 merges are less common than 1:m and m:1 merges.

merge 1:m and merge m:1 specify one-to-many and many-to-one match merges, respectively. To illustrate the two choices, suppose you have a dataset containing information about individual hospitals, called hospitals.dta. In this dataset, each observation contains information about one hospital, which is uniquely identified by the hospitalid variable. You have a second dataset called discharges.dta, which contains information on individual hospital stays by many different patients. discharges.dta also identifies hospitals by using the hospitalid variable. You would like to join all the information in both datasets. There are two ways you could do this.

`merge 1:m` *varlist* ... specifies a one-to-many match merge.

```
. use hospitals
. merge 1:m hospitalid using discharges
```

would join the discharge data to the hospital data. This is a `1:m` merge because `hospitalid` uniquely identifies individual observations in the dataset in memory (`hospitals`), but could correspond to many observations in the using dataset.

`merge m:1` *varlist* ... specifies a many-to-one match merge.

```
. use discharges
. merge m:1 hospitalid using hospitals
```

would join the hospital data to the discharge data. This is an `m:1` merge because `hospitalid` can correspond to many observations in the master dataset, but uniquely identifies individual observations in the using dataset.

`merge m:m` *varlist* ... specifies a many-to-many match merge. This is allowed for completeness, but it is difficult to imagine an example of when it would be useful. For an `m:m` merge, *varlist* does not uniquely identify the observations in either dataset. Matching is performed by combining observations with equal values of *varlist*; within matching values, the first observation in the master dataset is matched with the first matching observation in the using dataset; the second, with the second; and so on. If there is an unequal number of observations within a group, then the last observation of the shorter group is used repeatedly to match with subsequent observations of the longer group. Use of `merge m:m` is not encouraged.

`merge 1:1 _n` performs a sequential merge. `_n` is not a variable name; it is Stata syntax for observation number. A sequential merge performs a one-to-one merge on observation number. The first observation of the master dataset is matched with the first observation of the using dataset; the second, with the second; and so on. If there is an unequal number of observations, the remaining observations are unmatched. Sequential merges are dangerous, because they require you to rely on sort order to know that observations belong together. Use this merge at your own risk.

Basic description

Think of merge as being *master* + *using* = *merged result*.

Call the dataset in memory the *master* dataset, and the dataset on disk the *using* dataset. This way we have general names that are not dependent on individual datasets.

Suppose we have two datasets,

master in memory

id	age
1	22
2	56
5	17

on disk in file *filename*

id	wgt
1	130
2	180
4	110

We would like to join together the age and weight information. We notice that the `id` variable identifies unique observations in both datasets: if you tell me the `id` number, then I can tell you the one observation that contains information about that `id`. This is true for both the master and the using datasets.

Because `id` uniquely identifies observations in both datasets, this is a `1:1` merge. We can bring in the dataset from disk by typing

```
. merge 1:1 id using filename
```

in memory *master* + in *filename*.dta *using* = *merged result*

id	age
1	22
2	56
5	17

id	wgt
1	130
2	180
4	110

id	age	wgt	
1	22	130	(matched)
2	56	180	(matched)
5	17	.	(master only)
4	.	110	(using only)

The original data in memory are called the master data. The data in *filename*`.dta` are called the using data. After `merge`, the merged result is left in memory. The `id` variable is called the key variable. Stata jargon is that the datasets were merged on `id`.

Observations for `id==1` existed in both the master and using datasets and so were combined in the merged result. The same occurred for `id==2`. For `id==5` and `id==4`, however, no matches were found and thus each became a separate observation in the merged result. Thus each observation in the merged result came from one of three possible sources:

Numeric code	Equivalent word	Description
1	master	originally appeared in master only
2	using	originally appeared in using only
3	match	originally appeared in both

`merge` encodes this information into new variable `_merge`, which `merge` adds to the merged result:

in memory *master* + in *filename*.dta *using* = *merged result*

id	age
1	22
2	56
5	17

id	wgt
1	130
2	180
4	110

id	age	wgt	_merge
1	22	130	3
2	56	180	3
5	17	.	1
4	.	110	2

Note: Above we show the master and using data sorted by `id` before merging; this was for illustrative purposes. The dataset resulting from a `1:1` merge will have the same data, regardless of the sort order of the master and using datasets.

The formal definition for `merge` behavior is the following: Start with the first observation of the master. Find the corresponding observation in the using data, if there is one. Record the matched or unmatched result. Proceed to the next observation in the master dataset. When you finish working through the master dataset, work through unused observations from the using data. By default, unmatched observations are kept in the merged data, whether they come from the master dataset or the using dataset.

Remember this formal definition. It will serve you well.

1:1 merges

The example shown above is called a `1:1` merge, because the key variable uniquely identified each observation in each of the datasets.

A variable or variable list uniquely identifies the observations if each distinct value of the variable(s) corresponds to one observation in the dataset.

In some datasets, multiple variables are required to identify the observations. Imagine data obtained by observing patients at specific points in time so that variables `pid` and `time`, taken together, identify the observations. Below we have two such datasets and run a `1:1` merge on `pid` and `time`,

```
. merge 1:1 pid time using filename
```

master + *using* = *merged result*

pid	time	x1
14	1	0
14	2	0
14	4	0
16	1	1
16	2	1
17	1	0

pid	time	x2
14	1	7
14	2	9
16	1	2
16	2	3
17	1	5
17	2	2

pid	time	x1	x2	_merge
14	1	0	7	3
14	2	0	9	3
14	4	0	.	1
16	1	1	2	3
16	2	1	3	3
17	1	0	5	3
17	2	.	2	2

This is a `1:1` merge because the combination of the values of `pid` and `time` uniquely identifies observations in both datasets.

By default, there is nothing about a `1:1` merge that implies that all, or even any of, the observations match. Above five observations matched, one observation was only in the master (subject 14 at time 4), and another was only in the using (subject 17 at time 2).

m:1 merges

In an `m:1` merge, the key variable or variables uniquely identify the observations in the using data, but not necessarily in the master data. Suppose you had person-level data within regions and you wished to bring in regional data. Here is an example:

```
. merge m:1 region using filename
```

master + *using* = *merged result*

id	region	a
1	2	26
2	1	29
3	2	22
4	3	21
5	1	24
6	5	20

region	x
1	15
2	13
3	12
4	11

id	region	a	x	_merge
1	2	26	13	3
2	1	29	15	3
3	2	22	13	3
4	3	21	12	3
5	1	24	15	3
6	5	20	.	1
.	4	.	11	2

To bring in the regional information, we need to merge on `region`. The values of `region` identify individual observations in the using data, but it is not an identifier in the master data.

We show the merged dataset sorted by `id` because this makes it easier to see how the merged dataset was constructed. For each observation in the master data, `merge` finds the corresponding observation in the using data. `merge` combines the values of the variables in the using dataset to the observations in the master dataset.

1:m merges

1:m merges are similar to m:1, except that now the key variables identify unique observations in the master dataset. Any datasets that can be merged using an m:1 merge may be merged using a 1:m merge by reversing the roles of the master and using datasets. Here is the same example as used previously, with the master and using datasets reversed:

```
. merge 1:m region using filename
```

master + *using* = *merged result*

region	x
1	15
2	13
3	12
4	11

id	region	a
1	2	26
2	1	29
3	2	22
4	3	21
5	1	24
6	5	20

region	x	id	a	_merge
1	15	2	29	3
1	15	5	24	3
2	13	1	26	3
2	13	3	22	3
3	12	4	21	3
4	11	.	.	1
5	.	6	20	2

This merged result is identical to the merged result in the previous section, except for the sort order and the contents of _merge. This time, we show the merged result sorted by region rather than id. Reversing the roles of the files causes a reversal in the 1s and 2s for _merge: where _merge was previously 1, it is now 2, and vice versa. These exchanged _merge values reflect the reversed roles of the master and using data.

For each observation in the master data, merge found the corresponding observation(s) in the using data and then wrote down the matched or unmatched result. Once the master observations were exhausted, merge wrote down any observations from the using data that were never used.

m:m merges

m:m specifies a many-to-many merge and is a bad idea. In an m:m merge, observations are matched within equal values of the key variable(s), with the first observation being matched to the first; the second, to the second; and so on. If the master and using have an unequal number of observations within the group, then the last observation of the shorter group is used repeatedly to match with subsequent observations of the longer group. Thus m:m merges are dependent on the current sort order—something which should never happen.

Because m:m merges are such a bad idea, we are not going to show you an example. If you think that you need an m:m merge, then you probably need to work with your data so that you can use a 1:m or m:1 merge. Tips for this are given in *Troubleshooting m:m merges* below.

Sequential merges

In a *sequential* merge, there are no key variables. Observations are matched solely on their observation number:

```
. merge 1:1 _n using filename
```

master + *using* = *merged result*

x1
10
30
20
5

x2
7
2
1
9
3

x1	x2	_merge
10	7	3
30	2	3
20	1	3
5	9	3
.	3	2

In the example above, the using data are longer than the master, but that could be reversed. In most cases where sequential merges are appropriate, the datasets are expected to be of equal length, and you should type

```
. merge 1:1 _n using filename, assert(match) nogenerate
```

Sequential merges, like `m:m` merges, are dangerous. Both depend on the current sort order of the data.

Treatment of overlapping variables

When performing merges of any type, the master and using datasets may have variables in common other than the key variables. We will call such variables overlapping variables. For instance, if the variables in the master and using datasets are

master: `id`, `region`, `sex`, `age`, `race`
using: `id`, `sex`, `bp`, `race`

and `id` is the key variable, then the overlapping variables are `sex` and `race`.

By default, `merge` treats values from the master as inviolable. When observations match, it is the master's values of the overlapping variables that are recorded in the merged result.

If you specify the `update` option, however, then all missing values of overlapping variables in matched observations are replaced with values from the using data. Because of this new behavior, the merge codes change somewhat. Codes 1 and 2 keep their old meaning. Code 3 splits into codes 3, 4, and 5. Codes 3, 4, and 5 are filtered according to the following rules; the first applicable rule is used.

`5` corresponds to matched observations where at least one overlapping variable had conflicting nonmissing values.

`4` corresponds to matched observations where at least one missing value was updated, but there were no conflicting nonmissing values.

`3` means observations matched, and there were neither updated missing values nor conflicting nonmissing values.

If you specify both the `update` and `replace` options, then the `_merge==5` cases are updated with values from the using data.

Sort order

As we have mentioned, in the `1:1`, `1:m`, and `m:1` match merges, the sort orders of the master and using datasets do not affect the data in the merged dataset. This is not the case of `m:m`, which we recommend you never use.

Sorting is used by `merge` internally for efficiency, so the merged result can be produced most quickly when the master and using datasets are already sorted by the key variable(s) before merging. You are not required to have the dataset sorted before using `merge`, however, because `merge` will sort behind the scenes, if necessary. If the using dataset is not sorted, then a temporary copy is made and sorted to ensure that the current sort order on disk is not affected.

All this is to reassure you that 1) your datasets on disk will not be modified by `merge` and 2) despite the fact that our discussion has ignored sort issues, `merge` is, in fact, efficient behind the scenes.

It hardly makes any difference in run times, but if you know that the master and using data are already sorted by the key variable(s), then you can specify the `sorted` option. All that will be saved is the time `merge` would spend discovering that fact for itself.

The merged result produced by `merge` orders the variables and observations in a special and sometimes useful way. If you think of datasets as tables, then the columns for the new variables appear to the right of what was the master. If the master data originally had k variables, then the new variables will be the $(k+1)$st, $(k+2)$nd, and so on. The new observations are similarly ordered so that they all appear at the end of what was the master. If the master originally had N observations, then the new observations, if any, are the $(N+1)$st, $(N+2)$nd, and so on. Thus the original master data can be found from the merged result by extracting the first k variables and first N observations. If `merge` with the `update` option was specified, however, then be aware that the extracted master may have some updated values.

The merged result is unsorted except for a `1:1` merge, where there are only matched observations. Here the dataset is sorted by the key variables.

Troubleshooting m:m merges

First, if you think you need to perform an `m:m` merge, then we suspect you are wrong. If you would like to match every observation in the master to every observation in the using with the same values of the key variable(s), then you should be using `joinby`; see [D] **joinby**.

If you still want to use `merge`, then it is likely that you have forgotten one or more key variables that could be used to identify observations within groups. Perhaps you have panel data with 4 observations on each subject, and you are thinking that what you need to do is

```
. merge m:m subjectid using filename
```

Ask yourself if you have a variable that identifies observation within panel, such as a sequence number or a time. If you have, say, a `time` variable, then you probably should try something like

```
. merge 1:m subjectid time using filename
```

(You might need a `1:1` or `m:1` merge; `1:m` was arbitrarily chosen for the example.)

If you do not have a time or time-like variable, then ask yourself if there is a meaning to matching the first observations within subject, the second observations within subject, and so on. If so, then there is a concept of sequence within subject.

Suppose you do indeed have a sequence concept, but in your dataset it is recorded via the ordering of the observations. Here you are in a dangerous situation because any kind of sorting would lose the identity of the first, second, and nth observation within subject. Your first goal should be to fix this problem by creating an explicit sequence variable from the current ordering—your `merge` can come later.

Start with your master data. Type

```
. sort subjectid, stable
. by subjectid: gen seqnum = _n
```

Do not omit `sort`'s `stable` option. That is what will keep the observations in the same order within subject. Save the data. Perform these same three steps on your using data.

After fixing the datasets, you can now type

```
. merge 1:m subjectid seqnum using filename
```

If you do not think there is a meaning to being the first, second, and nth observation within subject, then you need to ask yourself what it means to match the first observations within `subjectid`, the second observations within `subjectid`, and so on. Would it make equal sense to match the first with the third, the second with the fourth, or any other haphazard matching? If so, then there is no real ordering, so there is no real meaning to merging. You are about to obtain a haphazard result; you need to rethink your merge.

Examples

▷ Example 1: A 1:1 merge

We have two datasets, one of which has information about the size of old automobiles, and the other of which has information about their expense:

```
. use http://www.stata-press.com/data/r13/autosize
(1978 Automobile Data)
. list
```

	make	weight	length
1.	Toyota Celica	2,410	174
2.	BMW 320i	2,650	177
3.	Cad. Seville	4,290	204
4.	Pont. Grand Prix	3,210	201
5.	Datsun 210	2,020	165
6.	Plym. Arrow	3,260	170

```
. use http://www.stata-press.com/data/r13/autoexpense
(1978 Automobile Data)
. list
```

	make	price	mpg
1.	Toyota Celica	5,899	18
2.	BMW 320i	9,735	25
3.	Cad. Seville	15,906	21
4.	Pont. Grand Prix	5,222	19
5.	Datsun 210	4,589	35

We can see that these datasets contain different information about nearly the same cars—the autosize file has one more car. We would like to get all the information about all the cars into one dataset.

Because we are adding new variables to old variables, this is a job for the merge command. We need only to decide what type of match merge we need.

Looking carefully at the datasets, we see that the make variable, which identifies the cars in each of the two datasets, also identifies individual observations within the datasets. What this means is that if you tell me the make of car, I can tell you the one observation that corresponds to that car. Because this is true for both datasets, we should use a 1:1 merge.

We will start with a clean slate to show the full process:

```
. use http://www.stata-press.com/data/r13/autosize
(1978 Automobile Data)
. merge 1:1 make using http://www.stata-press.com/data/r13/autoexpense
    Result                           # of obs.
    -----------------------------------------
    not matched                             1
        from master                         1  (_merge==1)
        from using                          0  (_merge==2)
    matched                                 5  (_merge==3)
    -----------------------------------------
. list
     +-------------------------------------------------------------------------+
     | make              weight   length    price   mpg            _merge      |
     |-------------------------------------------------------------------------|
  1. | BMW 320i           2,650      177    9,735    25      matched (3)       |
  2. | Cad. Seville       4,290      204   15,906    21      matched (3)       |
  3. | Datsun 210         2,020      165    4,589    35      matched (3)       |
  4. | Plym. Arrow        3,260      170        .     .   master only (1)       |
  5. | Pont. Grand Prix   3,210      201    5,222    19      matched (3)       |
     |-------------------------------------------------------------------------|
  6. | Toyota Celica      2,410      174    5,899    18      matched (3)       |
     +-------------------------------------------------------------------------+
```

The merge is successful—all the data are present in the combined dataset, even that from the one car that has only size information. If we wanted only those makes for which all information is present, it would be up to us to drop the observations for which _merge < 3.

◁

▷ Example 2: Requiring matches

Suppose we had the same setup as in the previous example, but we erroneously think that we have all the information on all the cars. We could tell merge that we expect only matches by using the assert option.

```
. use http://www.stata-press.com/data/r13/autosize, clear
(1978 Automobile Data)
. merge 1:1 make using http://www.stata-press.com/data/r13/autoexpense,
> assert(match)
merge:  after merge, not all observations matched
        (merged result left in memory)
r(9);
```

merge tells us that there is a problem with our assumption. To see how many mismatches there were, we can tabulate _merge:

```
. tabulate _merge
```

_merge	Freq.	Percent	Cum.
master only (1)	1	16.67	16.67
matched (3)	5	83.33	100.00
Total	6	100.00	

If we would like to list the problem observation, we can type

```
. list if _merge < 3
```

	make	weight	length	price	mpg	_merge
4.	Plym. Arrow	3,260	170	.	.	master only (1)

If we were convinced that all data should be complete in the two datasets, we would have to rectify the mismatch in the original datasets.

◁

▷ Example 3: Keeping just the matches

Once again, suppose that we had the same datasets as before, but this time we want the final dataset to have only those observations for which there is a match. We do not care if there are mismatches—all that is important are the complete observations. By using the `keep(match)` option, we will guarantee that this happens. Because we are keeping only those observations for which the key variable matches, there is no need to generate the `_merge` variable. We could do the following:

```
. use http://www.stata-press.com/data/r13/autosize, clear
(1978 Automobile Data)
. merge 1:1 make using http://www.stata-press.com/data/r13/autoexpense,
> keep(match) nogenerate
```

Result	# of obs.
not matched	0
matched	5

```
. list
```

	make	weight	length	price	mpg
1.	BMW 320i	2,650	177	9,735	25
2.	Cad. Seville	4,290	204	15,906	21
3.	Datsun 210	2,020	165	4,589	35
4.	Pont. Grand Prix	3,210	201	5,222	19
5.	Toyota Celica	2,410	174	5,899	18

◁

▷ Example 4: Many-to-one matches

We have two datasets: one has salespeople in regions and the other has regional data about sales. We would like to put all the information into one dataset. Here are the datasets:

```
. use http://www.stata-press.com/data/r13/sforce, clear
(Sales Force)
. list
```

	region	name
1.	N Cntrl	Krantz
2.	N Cntrl	Phipps
3.	N Cntrl	Willis
4.	NE	Ecklund
5.	NE	Franks
6.	South	Anderson
7.	South	Dubnoff
8.	South	Lee
9.	South	McNeil
10.	West	Charles
11.	West	Cobb
12.	West	Grant

```
. use http://www.stata-press.com/data/r13/dollars
(Regional Sales & Costs)
. list
```

	region	sales	cost
1.	N Cntrl	419,472	227,677
2.	NE	360,523	138,097
3.	South	532,399	330,499
4.	West	310,565	165,348

We can see that the region would be used to match observations in the two datasets, and this time we see that `region` identifies individual observations in the `dollars` dataset but not in the `sforce` dataset. This means we will have to use either an `m:1` or a `1:m` merge. Here we will open the `sforce` dataset and then `merge` the `dollars` dataset. This will be an `m:1` merge, because `region` does not identify individual observations in the dataset in memory but does identify them in the using dataset. Here is the command and its result:

```
. use http://www.stata-press.com/data/r13/sforce
(Sales Force)
. merge m:1 region using http://www.stata-press.com/data/r13/dollars
(label region already defined)
    Result                           # of obs.
    -----------------------------------------
    not matched                             0
    matched                                12  (_merge==3)
    -----------------------------------------
```

```
. list
```

	region	name	sales	cost	_merge
1.	N Cntrl	Krantz	419,472	227,677	matched (3)
2.	N Cntrl	Phipps	419,472	227,677	matched (3)
3.	N Cntrl	Willis	419,472	227,677	matched (3)
4.	NE	Ecklund	360,523	138,097	matched (3)
5.	NE	Franks	360,523	138,097	matched (3)
6.	South	Anderson	532,399	330,499	matched (3)
7.	South	Dubnoff	532,399	330,499	matched (3)
8.	South	Lee	532,399	330,499	matched (3)
9.	South	McNeil	532,399	330,499	matched (3)
10.	West	Charles	310,565	165,348	matched (3)
11.	West	Cobb	310,565	165,348	matched (3)
12.	West	Grant	310,565	165,348	matched (3)

We can see from the result that all the values of `region` were matched in both datasets. This is a rare occurrence in practice!

Had we had the `dollars` dataset in memory and merged in the `sforce` dataset, we would have done a `1:m` merge.

◁

We would now like to use a series of examples that shows how `merge` treats nonkey variables, which have the same names in the two datasets. We will call these "overlapping" variables.

▷ Example 5: Overlapping variables

Here are two datasets whose only purpose is for this illustration:

```
. use http://www.stata-press.com/data/r13/overlap1, clear
. list, sepby(id)
```

	id	seq	x1	x2
1.	1	1	1	1
2.	1	2	1	.
3.	1	3	1	2
4.	1	4	.	2
5.	2	1	.	1
6.	2	2	.	2
7.	2	3	1	1
8.	2	4	1	2
9.	2	5	.a	1
10.	2	6	.a	2
11.	3	1	.	.a
12.	3	2	.	1
13.	3	3	.	.
14.	3	4	.a	.a
15.	10	1	5	8

```
. use http://www.stata-press.com/data/r13/overlap2
```

```
. list
```

	id	bar	x1	x2
1.	1	11	1	1
2.	2	12	.	1
3.	3	14	.	.a
4.	20	18	1	1

We can see that `id` can be used as the key variable for putting the two datasets together. We can also see that there are two overlapping variables: `x1` and `x2`.

We will start with a simple `m:1` merge:

```
. use http://www.stata-press.com/data/r13/overlap1
. merge m:1 id using http://www.stata-press.com/data/r13/overlap2
    Result                           # of obs.
    -----------------------------------------
    not matched                             2
        from master                         1  (_merge==1)
        from using                          1  (_merge==2)
    matched                                14  (_merge==3)
    -----------------------------------------
. list, sepby(id)
```

	id	seq	x1	x2	bar	_merge
1.	1	1	1	1	11	matched (3)
2.	1	2	1	.	11	matched (3)
3.	1	3	1	2	11	matched (3)
4.	1	4	.	2	11	matched (3)
5.	2	1	.	1	12	matched (3)
6.	2	2	.	2	12	matched (3)
7.	2	3	1	1	12	matched (3)
8.	2	4	1	2	12	matched (3)
9.	2	5	.a	1	12	matched (3)
10.	2	6	.a	2	12	matched (3)
11.	3	1	.	.a	14	matched (3)
12.	3	2	.	1	14	matched (3)
13.	3	3	.	.	14	matched (3)
14.	3	4	.a	.a	14	matched (3)
15.	10	1	5	8	.	master only (1)
16.	20	.	1	1	18	using only (2)

Careful inspection shows that for the matched `id`, the values of `x1` and `x2` are still the values that were originally in the `overlap1` dataset. This is the default behavior of `merge`—the data in the master dataset is the authority and is kept intact.

◁

Example 6: Updating missing data

Now we would like to investigate the `update` option. Used by itself, it will replace missing values in the master dataset with values from the using dataset:

```
. use http://www.stata-press.com/data/r13/overlap1, clear
. merge m:1 id using http://www.stata-press.com/data/r13/overlap2, update
    Result                           # of obs.
    -----------------------------------------
    not matched                             2
        from master                         1  (_merge==1)
        from using                          1  (_merge==2)
    matched                                14
        not updated                         5  (_merge==3)
        missing updated                     4  (_merge==4)
        nonmissing conflict                 5  (_merge==5)
    -----------------------------------------
. list, sepby(id)
```

	id	seq	x1	x2	bar	_merge
1.	1	1	1	1	11	matched (3)
2.	1	2	1	1	11	missing updated (4)
3.	1	3	1	2	11	nonmissing conflict (5)
4.	1	4	1	2	11	nonmissing conflict (5)
5.	2	1	.	1	12	matched (3)
6.	2	2	.	2	12	nonmissing conflict (5)
7.	2	3	1	1	12	matched (3)
8.	2	4	1	2	12	nonmissing conflict (5)
9.	2	5	.	1	12	missing updated (4)
10.	2	6	.	2	12	nonmissing conflict (5)
11.	3	1	.	.a	14	matched (3)
12.	3	2	.	1	14	matched (3)
13.	3	3	.	.a	14	missing updated (4)
14.	3	4	.	.a	14	missing updated (4)
15.	10	1	5	8	.	master only (1)
16.	20	.	1	1	18	using only (2)

Looking through the resulting dataset observation by observation, we can see both what the `update` option updated as well as how the `_merge` variable gets its values.

The following is a listing that shows what is happening, where `x1_m` and `x2_m` come from the master dataset (`overlap1`), `x1_u` and `x2_u` come from the using dataset (`overlap2`), and `x1` and `x2` are the values that appear when using `merge` with the `update` option.

	id	x1_m	x1_u	x1	x2_m	x2_u	x2	_merge
1.	1	1	1	1	1	1	1	matched (3)
2.	1	1	1	1	.	1	1	missing updated (4)
3.	1	1	1	1	2	1	2	nonmissing conflict (5)
4.	1	.	1	1	2	1	2	nonmissing conflict (5)
5.	2	.	.	.	1	1	1	matched (3)
6.	2	.	.	.	2	1	2	nonmissing conflict (5)
7.	2	1	.	1	1	1	1	matched (3)
8.	2	1	.	1	2	1	2	nonmissing conflict (5)
9.	2	.a	.	.	1	1	1	missing updated (4)
10.	2	.a	.	.	2	1	2	nonmissing conflict (5)
11.	3	.	.	.	.a	.a	.a	matched (3)
12.	3	.	.	.	1	.a	1	matched (3)
13.	3	.	.	.	.	.a	.a	missing updated (4)
14.	3	.a	.	.	.a	.a	.a	missing updated (4)
15.	10	5	.	5	8	.	8	master only (1)
16.	20	.	1	1	.	1	1	using only (2)

From this, we can see two important facts: if there are both a conflict and an updated value, the value of `_merge` will reflect that there was a conflict, and missing values in the master dataset are updated by missing values in the using dataset.

◁

▷ Example 7: Updating all common observations

We would like to see what happens if the `update` and `replace` options are specified. The `replace` option extends the action of `update` to use nonmissing values of the using dataset to replace values in the master dataset. The values of `_merge` are unaffected by using both `update` and `replace`.

```
. use http://www.stata-press.com/data/r13/overlap1, clear
. merge m:1 id using http://www.stata-press.com/data/r13/overlap2, update replace
    Result                           # of obs.
    -----------------------------------------
    not matched                             2
        from master                         1  (_merge==1)
        from using                          1  (_merge==2)
    matched                                14
        not updated                         5  (_merge==3)
        missing updated                     4  (_merge==4)
        nonmissing conflict                 5  (_merge==5)
    -----------------------------------------
```

```
. list, sepby(id)
```

	id	seq	x1	x2	bar	_merge
1.	1	1	1	1	11	matched (3)
2.	1	2	1	1	11	missing updated (4)
3.	1	3	1	1	11	nonmissing conflict (5)
4.	1	4	1	1	11	nonmissing conflict (5)
5.	2	1	.	1	12	matched (3)
6.	2	2	.	1	12	nonmissing conflict (5)
7.	2	3	1	1	12	matched (3)
8.	2	4	1	1	12	nonmissing conflict (5)
9.	2	5	.	1	12	missing updated (4)
10.	2	6	.	1	12	nonmissing conflict (5)
11.	3	1	.	.a	14	matched (3)
12.	3	2	.	1	14	matched (3)
13.	3	3	.	.a	14	missing updated (4)
14.	3	4	.	.a	14	missing updated (4)
15.	10	1	5	8	.	master only (1)
16.	20	.	1	1	18	using only (2)

◁

▷ Example 8: More on the keep() option

Suppose we would like to use the `update` option, as we did above, but we would like to keep only those observations for which the value of the key variable, `id`, was found in both datasets. This will be more complicated than in our earlier example, because the `update` option splits the `matches` into `matches`, `match_updates`, and `match_conflicts`. We must either use all these code words in the `keep` option or use their numerical equivalents, 3, 4, and 5. Here the latter is simpler.

```
. use http://www.stata-press.com/data/r13/overlap1, clear
. merge m:1 id using http://www.stata-press.com/data/r13/overlap2, update
> keep(3 4 5)
```

Result	# of obs.	
not matched	0	
matched	14	
not updated	5	(_merge==3)
missing updated	4	(_merge==4)
nonmissing conflict	5	(_merge==5)

```
. list, sepby(id)
```

	id	seq	x1	x2	bar	_merge
1.	1	1	1	1	11	matched (3)
2.	1	2	1	1	11	missing updated (4)
3.	1	3	1	2	11	nonmissing conflict (5)
4.	1	4	1	2	11	nonmissing conflict (5)
5.	2	1	.	1	12	matched (3)
6.	2	2	.	2	12	nonmissing conflict (5)
7.	2	3	1	1	12	matched (3)
8.	2	4	1	2	12	nonmissing conflict (5)
9.	2	5	.	1	12	missing updated (4)
10.	2	6	.	2	12	nonmissing conflict (5)
11.	3	1	.	.a	14	matched (3)
12.	3	2	.	1	14	matched (3)
13.	3	3	.	.a	14	missing updated (4)
14.	3	4	.	.a	14	missing updated (4)

◁

▷ Example 9: A one-to-many merge

As a final example, we would like show one example of a `1:m` merge. There is nothing conceptually different here; what is interesting is the order of the observations in the final dataset:

```
. use http://www.stata-press.com/data/r13/overlap2, clear
. merge 1:m id using http://www.stata-press.com/data/r13/overlap1
```

Result	# of obs.	
not matched	2	
from master	1	(_merge==1)
from using	1	(_merge==2)
matched	14	(_merge==3)

```
. list, sepby(id)
```

	id	bar	x1	x2	seq	_merge
1.	1	11	1	1	1	matched (3)
2.	2	12	.	1	1	matched (3)
3.	3	14	.	.a	1	matched (3)
4.	20	18	1	1	.	master only (1)
5.	1	11	1	1	2	matched (3)
6.	1	11	1	1	3	matched (3)
7.	1	11	1	1	4	matched (3)
8.	2	12	.	1	2	matched (3)
9.	2	12	.	1	3	matched (3)
10.	2	12	.	1	4	matched (3)
11.	2	12	.	1	5	matched (3)
12.	2	12	.	1	6	matched (3)
13.	3	14	.	.a	2	matched (3)
14.	3	14	.	.a	3	matched (3)
15.	3	14	.	.a	4	matched (3)
16.	10	.	5	8	1	using only (2)

We can see here that the first four observations come from the master dataset, and all additional observations, whether matched or unmatched, come below these observations. This illustrates that the master dataset is always in the upper-left corner of the merged dataset.

◁

References

Golbe, D. L. 2010. Stata tip 83: Merging multilingual datasets. *Stata Journal* 10: 152–156.

Gould, W. W. 2011a. Merging data, part 1: Merges gone bad. The Stata Blog: Not Elsewhere Classified. http://blog.stata.com/2011/04/18/merging-data-part-1-merges-gone-bad/.

——. 2011b. Merging data, part 2: Multiple-key merges. The Stata Blog: Not Elsewhere Classified. http://blog.stata.com/2011/05/27/merging-data-part-2-multiple-key-merges/.

Nash, J. D. 1994. dm19: Merging raw data and dictionary files. *Stata Technical Bulletin* 20: 3–5. Reprinted in *Stata Technical Bulletin Reprints*, vol. 4, pp. 22–25. College Station, TX: Stata Press.

Weesie, J. 2000. dm75: Safe and easy matched merging. *Stata Technical Bulletin* 53: 6–17. Reprinted in *Stata Technical Bulletin Reprints*, vol. 9, pp. 62–77. College Station, TX: Stata Press.

Also see

[D] **append** — Append datasets

[D] **cross** — Form every pairwise combination of two datasets

[D] **joinby** — Form all pairwise combinations within groups

[D] **save** — Save Stata dataset

[D] **sort** — Sort data

[U] 22 Combining datasets