14

Data Filters

paste, cut, join, and uniq

14. Data Filters:sort, paste, cut, join, and uniq

There is a set of filters that perform data operations on an input stream. These data filters, receive data and generate modified output. The data filters are designed to operate on files whose text is organized into fields of data much like a single file database. Each line in the file is a record and each word in the line constitutes a field in that record. The data filters take as their input a file containing such records, and outputs records selected on the basis of a given criteria. These data filters can also be used to effectively manipulate any string of text. For example, you can use them to obtain filename prefixes.

There are five data filters: `sort`, `cut`, `paste`, `join`, and `uniq`. The `sort` filter generates a sorted version of the file in which all records are sorted alphabetically in ASCII order according to a specified field. `sort` is also a more general purpose filter that you can use to sort lines in any text file. The `cut` filter outputs all entries for a selected field in a data file. The `paste` filter generates output that combines the records of several data files. The `join` filter generates output that combines the records in two files by comparing the values of specified fields. The `uniq` filter detects fields that have the same values. It allows you to count how many fields have the same values as well as eliminating any repetitions from its output. This chapter will first examine the concept of Unix data files, and then discuss each of the data filters in turn.

Though the data filters cannot perform many of the complex operations found in professional database management software, you will find that they can perform many of the more common operations. You can sort data and selectively display fields. You can also selectively retrieve matching records in different files. You can even combined data filters to form complex queries. For example, you could use the `join` filter to combine selected records from different files and then pipe the output to the `sort` filter to sort the results.

UNIX Data Files

In Unix, a text file can be arranged in such a way that it can be interpreted as a data file. A data file consists of fields and records. Each record consists of a predetermined set of fields. The file itself consist of character text data, just like any other text file. However the character data is organized in a database format. Each line in the file constitutes a record. Each field is delimited on a line with either spaces, tabs, or a specially designated delimiter such as a colon. Data filters can then take the contents of such a file as input and generate modified output.

For example, a book data file may consist of records whose fields contain title, author, price, and publisher of a book. The records and fields can be represented in a two dimensional format. The fields are columns across the top of the data and records are rows of data. Below is an example of the data in a book data file.

```
               Fields
               Title          Author         Price
     Publisher
Records   1    Tempest        Shakespeare    15.75
     Penguin
          2    Christmas      Dickens        3.50
     Academic
          3    Iliad          Homer          10.25
     Random
          4    Raven          Poe            2.50
     Penguin
```

You can enter such data to a text file, making it a data file. Each line is a record. Each set of characters separated by a space or tab is a field in the record. Records can be typed in by the user with any text editor. Data operations such as `sort` and `cut` can then generate as output selected records or fields. In all cases the original files are left untouched. In the next example, a user has created a data file called **books**.

books

```
Tempest     Shakespeare 15.75  Penguin
Christmas   Dickens      3.50  Academic
Iliad       Homer       10.25  Random
Raven       Poe          2.50  Penguin
```

```
     $ cat books
     Tempest     Shakespeare  15.75  Penguin
     Christmas   Dickens       3.50  Academic
     Iliad       Homer        10.25  Random
     Raven       Poe           2.50  Penguin
```

Sort

As described in chapter 7, the `sort` filter outputs a sorted version of a file. `sort` is a very powerful utility with many different sorting options (see Table 14-1). These options are primarily designed to operate on files arranged in a data format. In fact, `sort` can be thought of as a powerful data manipulation tool, arranging records in a data file.

sorting lines: basic sort operations

The `sort` filter sorts character by character on a line. If the first characters in two lines are the same, `sort` will sort on the next characters. In the next example, the `sort` filter outputs a sorted version of perishables. Notice that the second and third lines begin with the same word, and differ on the first characters of the second words, vegetables and citris.

```
perishables
vegetable soup
fresh vegetables
fresh citris
lowfat milk
```

```
     $ sort perishables
     fresh citris
     fresh vegetables
     lowfat milk
     vegetable soup
```

You can, of course, save the sorted version in a file or send it to the printer.

```
     $ sort perishables > slist
     $ sort perishables | lp
```

You can use `sort` to sort the contents of several files. You list the files to be sorted as arguments to `sort`. In the next example, the contents of **perishables** and **packaged** are sorted into a combined list. This sorted version is then redirected to **plist**.

```
packaged
canned milk
frozen vegetabless
tomato paste
chocolate milk
```

```
     $ sort perishables packaged > plist
     $ cat plist
     canned milk
     chocolate milk
     fresh citris
     fresh vegetables
     frozen vegetables
     lowfat milk
     tomato paste
     vegetable soup
```

Since users usually want to save their sorted data in a file, the `sort` filter provides an option that designates a file in which to save its sorted output. The `-o` option followed by a filename will save the sorted output to that file. It performs the same function as redirection would. In the next example, the user sorts perishables and saves the sorted data in **slist** using the `-o` option.

```
$ sort perishables -o slist
$ cat slist
fresh citris
fresh vegetables
lowfat milk
vegetable soup
```

You may, at times, want to actually sort your original input file. This involves modifying your input file. To do this you may be tempted to redirect the output of `sort` to your original input file. This would be a mistake. Remember that the redirection operator executes before any Unix command, destroying a file if it already exists. In the command `sort perishables > perishables`, the redirection operator will first erase the perishables file to prepare it to receive input, and then the `sort` filter will attempt to read the just erased perishables file.

If you want to modify your input file, overwriting it with a sorted version of itself, then you can use the `-o` option. In the next example, the `sort` filter sorts the **perishables** file and then overwrites the **perishables** file with the sorted data.

```
$ sort perishables -o perishables
$ cat perishables
fresh citris
fresh vegetables
lowfat milk
vegetable soup
```

sorting data

By default the `sort` filter sorts data in alphabetic order. This alphabetic order is determined by the character set currently used by the system. This is usually the ASCII character set. The character set determines the sequence in which characters are ordered. A distinction is made between upper and lower case. The character set also has individual numbers, 0-9, which are considered characters. A number is treated as a sequence of individual numeric characters.

Such an ordering of the character set can lead to unintentional sorting results. Suppose you are mixing upper and lower case characters in your sort. Then any lines beginning with uppercase characters will be placed at the beginning of the sorted list because uppercase characters come before lowercase characters in the character set. The `sort` filter has several options that allow you to overcome the limitations of the character set. You can sort regardless of case, or sort numbers based on their numeric value. You can even sort in reverse order.

Ignoring case: -f

In the character set, the upper and lower case of a character are different characters. 'A' and 'a' are two distinct characters. All upper case characters come before lowercase characters in the character set ordering. `sort` will always place any uppercase character before a lowercase character. 'Z' will come before 'a'. For example, in the ASCII character set characters are represented by a sequence of integer numbers from 0 to 255. The numbers representing lowercase characters are from 97 to 122, whereas uppercase characters are 65 to 90. A lowercase 'b', 98, will always be greater than an uppercase 'B', 66. In the next example, the **perishablesU** file has several uppercase characters at the beginning of a line. A sort will place the lines beginning with uppercase characters at the top of the sorted list.

perishablesU

```
Vegetable soup
fresh vegetables
Fresh citris
lowfat milk
```

```
    $ sort perishablesU
    Fresh citris
    Vegetable soup
    fresh vegetables
    lowfat milk
```

With the `-f` option you can instruct `sort` to ignore any case differences. A lowercase 'a' and an uppercase 'A' will have the same place in the sorted data. 'a' will come before 'Z' and 'C' after 'b'. The `-f` option is called the fold option because it actually folds lowercase characters into uppercase, treating the whole file as if it were written in uppercase. In the next example the `sort` filter with the `-f` option ignores any upper or lower case distinctions in performing the sort.

```
    $ sort -f perishablesU
    Fresh citris
    fresh vegetables
    lowfat milk
    Vegetable soup
```

Sorting numbers: -n

In a character set, numbers are interpreted as individual characters and compared accordingly. In the next example, each line in the **myitems** file begins with the number of items needed. A sort on the list will sort on the numbers as characters, not as numeric values. The character '1' comes before the character '3' which comes before '4', and then '8'.

```
myitems
8 vegetable soup
12 fresh vegetables
45 fresh citris
3 lowfat milk
```

```
$ sort myitems
     12 fresh vegetables
     3 lowfat milk
     45 fresh citris
     8 vegetable soup
```

The `sort` filter has an option, -n, that allows you to sort numbers according to their numeric value. Negative numbers are sorted accordingly. With the -n option, the number 12 will be placed after the number 8. In the next example, the numbers are sorted by their numeric value.

```
     $ sort -n myitems
     3 lowfat milk
     8 vegetable soup
     12 fresh vegetables
     45 fresh citris
```

As an example, suppose that you want to find the largest file in your directory. The output of `ls -s` list all files and their sizes by blocks. The size is the first field. You could filter this output through a `sort` filter with the `-n` command to place the filename of the largest file at the end of the output, and then use the tail filter with the `-1` option to just display that filename.

```
     $ ls -l | sort -n | tail -1
```

Reverse sorts: -r

By default, `sort` orders lines in ascending order, from the lesser value to the greater value such as 'a' to 'z'. However, with the `-r` option, you can sort lines in descending order, starting from greater to lesser values. The `-r` option stands for reverse. Alphabetic values will be sorted beginning with 'z' and descending to 'a'. Numeric value will begin with the largest number and descend toward zero. In the next example, the **perishablesU** file is sorted in reverse order. Notice the combined use of both the `-r` and `-f` option, `-rf`.

```
     $ sort -rf perishablesU
     Vegetable soup
     lowfat milk
     fresh vegetables
     Fresh citris
```

In a numeric sort, the `-r` option will display the largest number first. In the next example, the **myitems** file is sorted numerically in reverse order.

```
      $ sort -rn myitems
      45 fresh citris
      12 fresh vegetables
      8 vegetable soup
      3 lowfat milk
```

Sorting Fields

As previously noted, a file can be organized with a data format. Each line is a record and each word is a field in the record. In the next example, the **listdata** file organizes each record into four fields. The first field is the number of items, the second field is the food type, the third field is the food name, and the fourth field is the price. Each field is separated by a space that acts as a delimiter between fields.

With selected fields in a file you can form a key with which the `sort` filter can sort the file. The key can be any sequence of fields. Each field is number beginning from 1. You specify a key using a combination of the +num option for the first field in the key and the -num option for the last field in the key. The `+` option is slightly odd in that the number you specify with it, is actually the number of fields you skip before the first field in the key. A `+2` means the key begins with the 3rd field, not the 2nd field. The first 2 fields are skipped. However, the number that you specify with the - option is the last field in the key. A `-4` means that the 4th field is the key's last field. In the next example and in figure 14.1, the user specifies a key consisting of only the second field using the options: `+1 -2`. The `+1` option skips the first field to begin the key at the 2nd field. The `-2` option ends the key at the second field. In this way, the user sorts only on the one field, ignoring the fields before and after it.

```
listdata
8 vegetable soup 1.75
12 fresh vegetables 7.50
45 fresh citris 3.00
3 lowfat milk 2.90
```

```
      $ sort +1 -2 listdata
      12 fresh vegetables  7.50
      45 fresh citris 3.00
      3  lowfat milk 2.90
      8  vegetable soup 1.75
```

Should you want to sort on just the first field you can specify the `+` option of the key as `+0`. This means that zero number of fields are skipped, so that the key begins with the first field. The `-1` option ends the key at the first field. The file will then be sorted only on the first field. Each key can be qualified by its own sort option. In the next example, the key specifies the first field and qualifies it as a numeric sort with the n option: `+0n -1`.

```
$ sort +0n -1 listdata
3  lowfat milk 2.90
8  vegetable soup 1.75
12 fresh vegetables  7.50
45 fresh citris 3.00
```

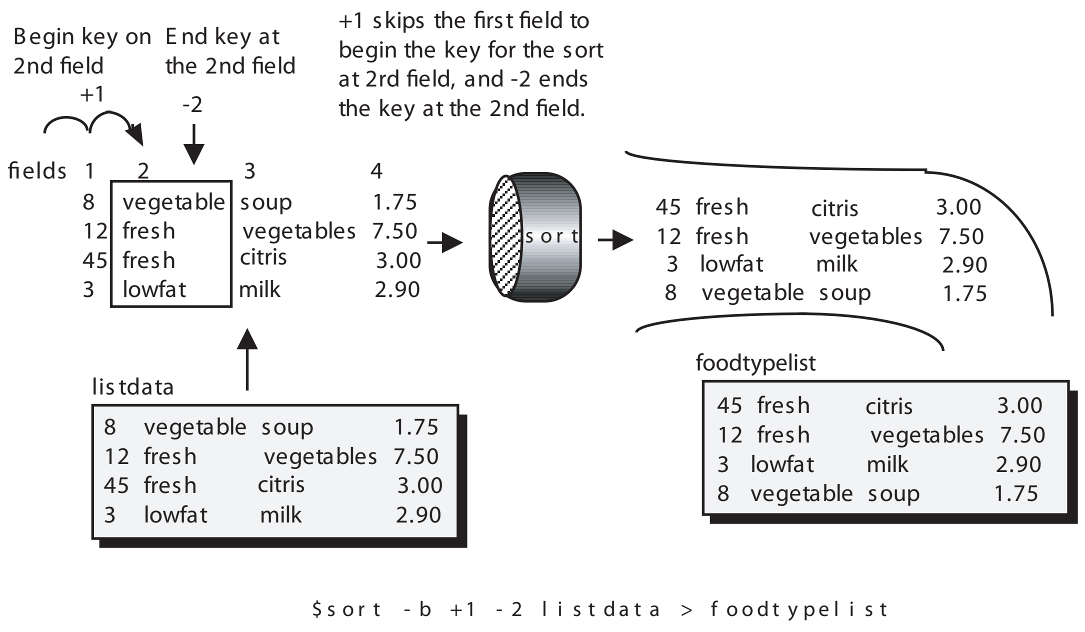


Figure 14.1. Sorting on a single field using a key specification consisting of the + and - options.

If you do not specify the last field in a key, then `sort` assumes that the remaining fields in the line form the key. The `+2` option used without any –*num* option, will include all fields from 3 to the end as part of the key. Used in this way, you can think of the +*num* option as a fields skipper, skipping over initial fields and sorting on the remainder of the line. Notice that you specify a particular field with a number that is one less than its own position. `+1` skips the first field and starts the sort on the 2nd field. `+3` skips the first three fields and starts the sort on the 4th field. In the next example, the user sorts the **listdata** file beginning with the third field. To do this the user must skip the first 2 fields.

```
     $ sort +2 listdata
     45 fresh citris 3.00
     3 lowfat milk 2.90
     8 vegetable soup 1.75
     12 fresh vegetables 7.50
```

To sort the last field you need only to skip the preceding fields. In the next example the user skips the first three fields to sort on the last field.

```
     $ sort -n +3 listdata
     8 vegetable soup 1.75
     3 lowfat milk 2.90
     45 fresh citris 3.00
     12 fresh vegetables 7.50
```

For example, suppose you want to list your files sorted by month, listing all those updated in the same month together. The `ls -l` command output full data about each file including its date. The month field would be the fifth field for with you use the option `+4`. The command `ls -l | sort +4` then lists your files sorted by month.

```
     $ ls -l | sort +4
```

When working with fields, you need to pay special attention to delimiters. In the **listdata** file example single spaces were used as delimiters. The space is the default delimiter for the `sort` filter. However, often a file will be arranged so that the data values in each field line up. You can do this either by using a tab instead of a space to separate fields, or by adding in leading spaces to line up the fields. In the **listdataS** file displayed below, the different fields are lined up by adding leading spaces. However, in the case of leading spaces, the `sort` filter will take one space as a field separator, but include the others as part of the field value and sort on them. In the character set, a space is always less than any alphabetic character. This means that fields with leading spaces will be ordered according to the number of spaces, beginning with the field with the most spaces. In the next example, the user sorts on the third field, skipping the first two fields, but including leading spaces. Notice that 'vegetables' is ranked second where it should be ranked last.

```
listdataS
8  vegetable soup        1.75
12 fresh     vegetables  7.50
45 fresh     citris       3.00
3  lowfat    milk        2.90
```

```
$ sort +2 -3 listdataS
45 fresh     citris       3.00
12 fresh     vegetables  7.50
3  lowfat    milk        2.90
8  vegetable soup        1.75
```

You can overcome the problem of leading spaces by using the `-b` option. The `-b` option instructs `sort` to ignore leading spaces in any specified key fields. On some recent versions of Unix the `-b` option is a default for any field key specification and will not have to be added. In such versions, leading spaces are automatically removed when fields are sorted. In the next example, the `sort` filter with the `-b` option, sorts the data on the third field ignoring leading spaces.

```
$ sort -b +2 -3 listdataS
45 fresh     citris       3.00
3  lowfat    milk        2.90
8  vegetable soup        1.75
12 fresh     vegetables  7.50
```

You can also enter the `-b` option as a qualifier to a specific key. The `+2b -3` key specification applies the `-b` option only to the third field.

Subsorts

You can instruct `sort` to perform several sub-sorts by using several key specifications. With each key, `sort` performs a sub-sort on that field. For example, you could sort **listdata** by food type and then by price using the key specifications `+1 -2` and `+3 -4`. First the list will be ordered by food type and then any food types that are the same are then ordered by price. Both 'citris' and 'vegetables' have a food type of 'fresh'. They would be further sorted by price. In the example, the user performs the subsort on the fourth field. Notice that the fourth field is further qualified as a numeric field with the n option.

```
$ sort -b +1 -2 +3n -4 listdataS
45 fresh     citris       3.00
12 fresh     vegetables  7.50
3  lowfat    milk        2.90
8  vegetable soup        1.75
```

You need to take into account that a subsort is only performed on those records whose compared fields in the primary sort were the same. If you were to use just `+1` as the key for the first sort, then `sort` compares the line beginning with the second field and including all the others. Since the rest of the lines are not the same, no subsort is performed. In this case, for a subsort to be performed on any lines, their second, third, and fourth fields would have to be the same.

Character Keys

Though a key consists of fields, you can, within those fields skip a number of characters so that the key actually begins further on in the field instead of at its beginning. The number of characters to be skipped is specified with a preceding period and is attached to the either the `+` or `–` options. `+2.4` begins the key at the 5th character in the third field. The first 2 fields are skipped, and then within the 3rd field, the first 4 characters are skipped. The + option can be said to have the format `+f.`*c* where f is the number of the fields to be skipped, and *c* is the number of characters to be skipped in the selected field. The same is true for the – option.

As an example, suppose that you want to skip the first two character in the fourth field of the **listdataS** file, and sort only on the cents and not the dollar value. Your key would have to skip the first two characters in the key field. The key specification would be `+3.2nb -4`. Skip the first 3 fields and then the first 2 character in the 4th field. Notice that the two option qualifiers, `n` and `b`, are added at the end of the character specification.

```
     $ sort +3.2nb -4 listdataS
     45 fresh     citris       3.00
     12 fresh     vegetables  7.50
     8  vegetable soup        1.75
     3  lowfat    milk        2.90
```

Field Delimiters

You could also use a specific delimiter such as a tab or a colon for each field, instead of spaces. If you should use such a designated delimiter, you could then include spaces in your fields, giving you fields consisting of several words. However, if you use your own delimiter, you need to specify it on the command line using the `-t` option. The `-t` option is followed immediately by a character that is then used by `sort` as the field delimiter for the input. In the next example, the **booksC** file uses the colon as the delimiter for each field. The user then sorts on the first field, the book titles, and specifies the field delimiter with a `-t` options, `-t:`.

```
booksC
War and Peace:Tolstoy:15.75:Penguin
Christmas carol:Dickens:3.50:Academic
Iliad:Homer:10.25:Random
Raven:Poe:2.50:Penguin
```

```
     $ sort -t: booksC
     Christmas carol:Dickens:3.50:Academic
     Iliad:Homer:10.25:Random
     Raven:Poe:2.50:Penguin
     War and Peace:Tolstoy:15.75:Penguin
```

You could also use a tab as a field delimiter. However, when you specify a tab as the character used for the `-t` option, you need to first quote it with a backslash. A tab on the command line is normally interpreted by the shell as just another space separating arguments. You first enter the `-t`, a backslash, and then hit the tab key. The tab character itself does not show up. In the next example, the **booksTD** file uses tabs to delimit each field. The user then sorts the file based on the second field, the authors. The field delimiter is specified with the option: `-t\tab`. The tab, of course does not show.

```
booksTD
War and Peace     Tolstoy   15.75  Penguin
Christmas carol  Dickens    3.50  Academic
Iliad            Homer     10.25  Random
Raven            Poe        2.50  Penguin
```

```
     $ sort -t\      +1 -2 booksTD
     Christmas carol  Dickens   3.50  Academic
     Iliad            Homer    10.25  Random
     Raven            Poe       2.50  Penguin
     War and Peace    Tolstoy  15.75  Penguin
```

TABLE 14-1

The `sort` filter:

`sort`

The `sort` filter sorts the lines it receives as input. You use it to generate a sorted version of a file. You can sort in a variety of ways such as alphabetic sorts, reverse sorts, and numeric sorts. You can sort on a given field or range of fields. The syntax for the `sort` filter is the keyword `sort` followed by any options and then a list of file names. `sort` can also receive its input from the standard input. You can pipe data into `sort` to be sorted.

```
$ sort -option file-list
$ sort -n perishables
```

basic `sort` operations

`-o` *filename*	Save the output of `sort` in filename. You can use this option to safely overwrite the original input file, giving you a sorted file. `$ sort perishables -o perishables`
c	Check only to see of the file is sorted. If the file is not sorted, `sort` displays an error message. Otherwise it displays nothing.
`m`	Merge previously sorted files.
`u`	Output repeated line only once.

sorting data

`d` Dictionary sort - ignores any characters in the character set that are not alphabetic, numbers, or blanks. Punctuation characters and control characters are ignored.

`f` Ignore case. Lowercase characters are folded into uppercase characters.

```
      $ sort -f perishables
```

`i` Ignore non-printing characters.

`M` Sort months. Fields whose values are the names of the month are sorted. The first three characters of the name are examined and changed to uppercase for sorting: JAN, FEB, JUN, NOV. They are order according to the months of the year beginning with January.

`n` Numeric sort - sort according to the numeric value of a field, not its character value. The -b option is automatically applied, ignoring any leading blanks.

```
      $ sort -n perishables
```

`-r` Sort in reverse order.

```
      $ sort -r perishables
```

sorting fields

`b` Ignore any leading blanks before a field.

```
      $ sort -b perishables
```

+*num* The number of fields to skip on a line. Sorting begins from the next field. +2 skips the first two fields and begins sorting on the third field.

```
      $ sort +2 perishables
```

-*num* The number of field where sorting on a line ends. `-3` will stop a sort on a line at the third field. Fields after the third field would not be used in the sort. This option is often used in conjunction with the + option to isolate a field, and restrict the sort to that field. `+2 -3` sorts only on the third field. You can also specify a range of fields: +1 -4.

```
      $ sort +2 -3 perishables
```

`-tc` Specify a new field delimiter, c. The default is a space.

```
      $ sort -t: +2  booksC
      $ sort -t\tab   +2  booksT
```

Paste: combining records

The `paste` command generates output that joins each line in different files into one line. By default, the lines from each file are placed on the same line, separated by a tab. In regards to data files, you can think of `paste` as joining the records in different files into one record. In effect, you add new fields to a record. You can also think of `paste` as joining files together side by side, each file contributing its own set of columns. In the next example and in figure 14.2, the lines of the files foods and costs are joined together in the output. The output is redirected to the **mylist** file.

The `cut` filter references a specific field using the `-f` option and a number. The `-f` option instructs `cut` what fields to copy. The `-f` option is followed by the number of the field you want. All fields are automatically numbered, beginning with `1`. There is no default. Whenever you use `cut` you need to have either a `-f` or `-c` option, specifying the column of data that you want to operate on. Furthermore, the `cut` filter with the `-f` option assumes that fields in your file are delimited with tabs, not spaces. `-f` and `-c` along with other `cut` options are listed in Table 14-3. In the next example, the **listdataD** file has its fields separated by tabs. The `cut` filter then copies out the second field in the **listdataD** file.

`listdatadD`
```
8  vegetable soup        1.75
12 fresh     vegetables  7.50
45 fresh     citris       3.00
3  lowfat    milk        2.90
```

```
     $ cut -f2 listdataD
     vegetable
     fresh
     fresh
     lowfat
```

You can copy several fields by listing their field numbers separated by commas after the `-f` option. You can even reference a range of fields using the first field number and the last number separated by a minus sign. In the next example and in figure 14.3, the fields from 1 and 3 are output.

```
     $ cut -f1,3 listdataD
     8  soup
     12 vegetables
     45 citris
     3  milk
```

In the next example, the user specifies a range for fields 2, 3, and 4: 2-4.

```
     $ cut -f2-4 listdataD
     vegetable soup        1.75
     fresh     vegetables  7.50
     fresh     citris       3.00
     lowfat    milk        2.90
```

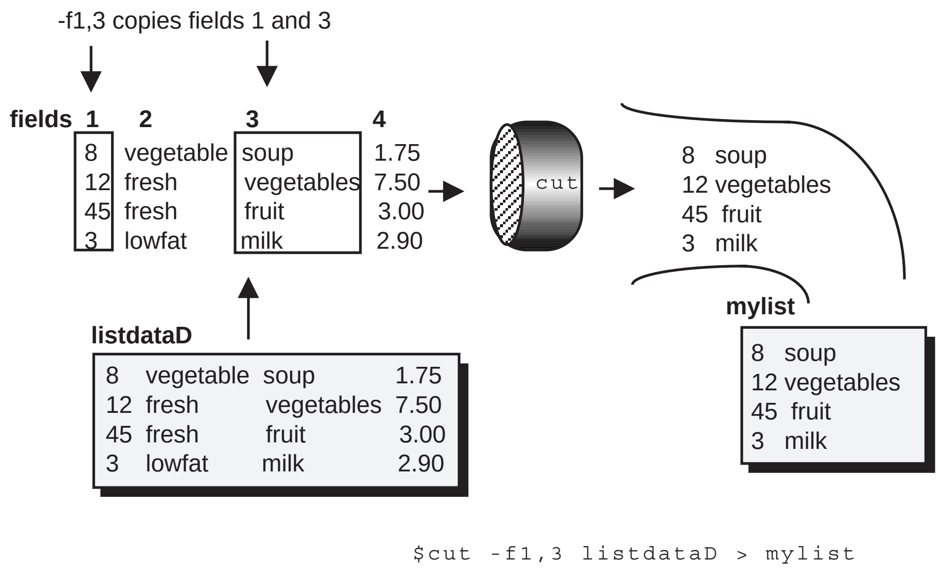


Figure 14.3. The cut filter.

Data files can be arranged in a fixed file format or a delimited file format. A delimited file format has its field separated by a designated delimiter. The **listdataD** file has been delimited with tabs. But it could also be delimited with any other character you choose. Often the delimiter is a character that is not often used such as a comma or a colon. Delimited files have the advantage of allowing field values of any length. A delimiter always tells where a field ends. By default, the `cut` filter will assume a tab delimiter in the file. If your file uses a different delimiter, you have to specify that delimiter with the `-d` option. For example, if your delimiter is a colon you need to include the option `-d:`. If your fields are separated by spaces, you need to specify a space as the delimiter: `-d' '`. Be sure to quote the space so that it will not be interpreted by the shell. In the next example the **listdataC** file is delimited with colons. The user then copies the first and third fields, and specifies that the colon is the delimiter used in this file.

listdataC

```
8:vegetable:soup:1.75
12:fresh:vegetables:7.50
45:fresh:citris:3.00
3:lowfat:milk:2.90
```

```
$ cut -f1,3 -d: listdataC
8:soup
12:vegetables
45:citris
3:milk
```

You can combine the cut filter with other filters to output selected fields of data. For example, the ls -l command outputs full information about all the files in your current directory, beginning with permissions and including links, size, date as well as filenames. Suppose you only want to list permissions and filenames. You could filter the output of ls -l through a cut filter to select only the fields for permissions and filenames. If you look at the output of the ls -l command you will notice a data format with each field seperated by spaces. The data field for filenames is the 8th and last field. The cut command cut -f1,8 will then output only the permissions and filenames, the 1st and 8th field. Remember, however, that the cut command operates on fields seperated by tabs. You need to first replace the seperating spaces in the ls -l output to tabs before having cut operate on it. This you can do using a sed filter with the editing command 's/spacespace*/tab/g' (the space and tab characters will not show so the command will appear as: 's/ */   /g'). There are two spaces entered before the asterisk indicating a match on one or more spaces. The next example shows the

```
$ ls -l | sed 's/  */    /g' | cut -f1,8
```

Many data files have a fixed length format. In this format, each field takes up a fixed number of characters in a line. In the **listdataS** file, fields are separated by spaces. The first field takes up three characters. The third field takes up 11 characters. Each field begins at a fixed position on the line. In the **listdataS** file, the second field always begins at the 4th character, and the fourth field begins at the 25th character. Taking this information into consideration, you can effectively reference a field using the beginning and ending character of the field on the line.

Using the -c option, the cut filter can reference character positions in a fixed length format file. The -c option can reference characters on a line, rather than a field. -c stands for column and takes a number or set of numbers as its argument. The numbers refer to character columns. Instead of looking at a file horizontally, line by line, you can look at it vertically, by columns of characters. The fourth column in the mylist file consists of the characters 'v', 'f', 'f', and 'l'. If you want to reference the second field, you can specify a range of columns beginning with 4 and ending with 13: -c4-13. You can think of the numbers used with -c as representing a position of a character on a line. -c4-13 references characters beginning at the 4th position through to the 13th position. In the next example, the user copies out the first and third fields, referencing by the range of their character columns.

```
listdataS
8  vegetable soup        1.75
12 fresh     vegetables  7.50
45 fresh     citris       3.00
3  lowfat    milk        2.90
```

```
$ cut -c1-3,14-24 listdataS
8  soup
12 vegetables
45 citris
3  milk
```

A number and a minus sign alone reference the rest of the line beginning with the character at that position. `-c14-` references all characters from the 14th position to the end of the line. In the next example, the user simply references the rest of the line beginning the character at position 14. This happens to be the beginning of the third field.

```
     $ cut -c14- listdataS
     soup        1.75
     vegetables  7.50
     citris       3.00
     milk        2.90
```

The output of the `ls -l` command also has a fixed length format. To output the permission and name fields you could use the specification `c1-10,c46-`. The permissions field begins with the 1st character and ends at the 10th. The filenames field begins at the 46th character and continues to the end of the line. Notice that by using the character specification there is no need to insert tab delimiters.

```
     $ ls -l | cut -c1-10,46-
```

TABLE 14-3

The cut filter:

`cut`

The `cut` filter copies out specified fields or columns in a file. You must always use either the `-f` option or the `-c` option with `cut`.

```
$ cut -option file-list
$ cut -f2,3 listdataD
```

`-f`*num*	The `-f` option specifies what fields you want copied out of a file. Fields are numbers from 1. You can specify more than one field by separating them with a comma, or you can specify a range of fields using a dash between numbers. *-fnum1*`,`*num2* Specify fields to be cut out.. `$ cut -f1,3 listdataD` *-fnum1*`-`*num2* Specify a range of fields beginning with num1 and ending with num2. `$ cut -f2-4 listdataD`
`-c`*num*`-`*num*	The `-c` option allows you to specify columns of characters to be cut out. `$ cut -c20-35 listdataS`
`-d`*delimiter-list*	The `-d` option allows you to specify your own delimiter to look for in a file. `$ cut -d: -f2-4  listdataC`
`-s`	Ignores any lines that do not have a delimiter in them. This option can only be used with the `-f` *option.* You use it to pass over lines with no data, such as headings, titles, or empty lines. `$ cut -f2-4 -s listdataD`

Join: Comparing Fields

The `join` filter compares the values of a designated field in one file with that of a field in another file. If the fields in each file have the same value, then the lines in each file are joined into one line. In effect, `join` is performing a conditional paste. Only those lines whose designated fields match, will be joined and output. The fields that you compare in each file must meet one condition before `join` can effectively operate. They must be sorted.

You use the `-j` option to specify what field you want compared. The `-j` option takes as its argument the number of the field to be joined in each file. There is a space between the `-j` option and the field number. As in the `paste` and `cut` filters, fields are numbered from 1. Table 14-4 list the `-j` option along with other `join` options.

```
    $ join -j fieldnum filelist
```

In the next example the lines in two files are selected and combined if their first field matches. To simplify the example, the first fields are already sorted. The **foodlist** file consists of three fields: items, food types, and price. The counts file consists of two fields: items and the number of each item. Notice that both the **foodlist** and counts files have an items field. The items field in each will be compared and used to join lines in the two files.

```
foodlist
citris         fresh      3.00
milk          lowfat    2.90
milk          canned    1.50
soup          vegetable 1.75
vegetables    fresh     7.50
```

```
counts
milk          3
soup          8
vegetables    12
```

```
     $ join -j 1 foodlist counts > mylist
     $ cat mylist
     milk         lowfat    2.90   3
     milk         canned    1.50   3
     soup         vegetable 1.75   8
     vegetables   fresh     7.50   12
```

Usually the fields being compared hold the same type of data. For example, the item field in one file can be compared with the item field in another file. Such a field is often referred to as a key field. It is a field with which you can connect the data in one field with another. However, two fields of the same type may have different field positions in their respective files. The item field in one file may be field 1 and the item field in another file could be field 2. In order to compare fields that have different positions, you need to include a separate `-j` option and field number for each file. To specify what file a given `-j` option refers to you need to also include a file number.   The file number refers to the position of the file name in the file list. The `-j` option can take a file number as an argument. The file number is placed right next to the `-j` on the command line. There is, of course, a space between the file number and the field number. The option `-j2 3` references the second file listed in the command line and the third field in that file. Each file may have its own `-j` option.

In the next example, the **foodtypes** file is a re-arranged version of the foods file in which items is the second field and the food types the first field. The `-j` options are then used to reference the items field(1 2) in the **foodtypes** file and compare it to the items field(2 1) in the counts file.

```
foodtypes
fresh     citris      3.00
lowfat    milk       2.90
canned    milk       1.50
vegetable soup       1.75
fresh     vegetables  7.50
```

```
counts
milk        3
soup        8
vegetables  12
```

```
     $ join -j1 2 -j2 1 foodtypes counts > mylist
     $ cat mylist
     milk        lowfat    2.90   3
     milk        canned    1.50   3
     soup        vegetable 1.75   8
     vegetables  fresh     7.50   12
```

If no `-j` option is used in the `join` filter then the first fields in each file are compared by default. The three example below are equivalent. All would compare the first fields of each file.

```
     $ join foodlist counts
     $ join -j 1 foodlist counts
     $ join -j1 1 -j2 1 foodlist counts
```

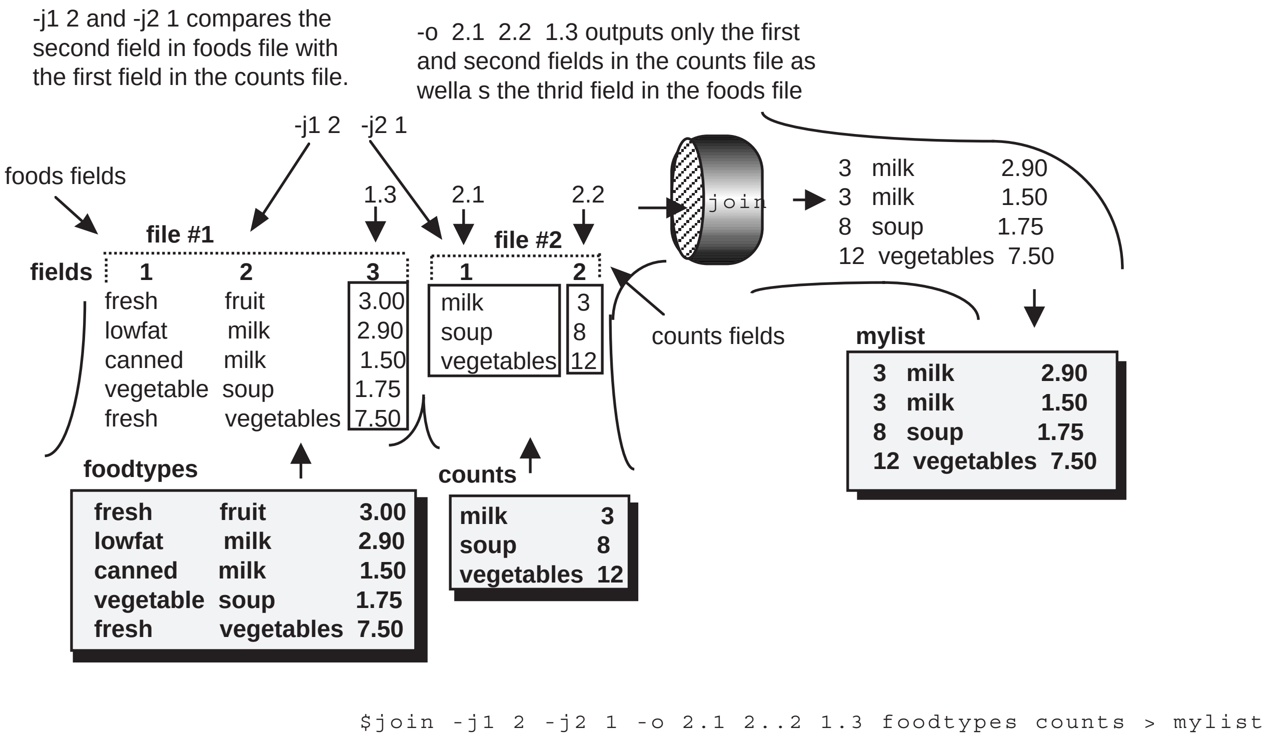
Selecting Output Fields: -o

By default, the `join` command will output all the fields in both files. However, the `-o` option lets you select what fields you want to output. It operates somewhat like the `cut` filter, only outputting specified fields rather than the whole line. The `-o` option takes as its argument a group of file and field numbers. A file number and a field number are connected by a period. The file number references a specific file and the field number specifies what field in the file is to be output. The file and field number `2.4` references the fourth field in the second file. You can list several file and field numbers, separating each pair with a space. The group of file and field number `2.4 1.1` references the fourth field in the second file and the first field in the first file. In the next example and in figure 14.4, the first field in the second file (`2.1`), the second field of the second file (`2.2`), and the third field in the first file (`1.3`) are output as a result of comparing the second field in **foodtypes** and the first field in counts.

```
     $ join -j1 2 -j2 1 -o 2.1 2.2 1.3 foodtypes counts >
mylist
     $ cat mylist
     milk        3  2.90
     milk        3  1.50
     soup        8  1.75
     vegetables  12 7.50
```

You can combine `join` with other data filters to form more complex queries. In the next example, the output from the previous query is piped into `sort` which then sorts the lines according to the most expensive prices. The `+2` option to sort skips the first two fields and the nr options sorts the third field by number in reverse order.

```
     $ join -j1 2 -j2 1 -o 2.1 2.2 1.3 foodtypes counts |
sort +2nr
     vegetables  12 7.50
     milk        3  2.90
     soup        8  1.75
     milk        3  1.50
```



-j1 2 and -j2 1 compares the second field in foods file with the first field in the counts file.
-o 2.1 2.2 1.3 outputs only the first and second fields in the counts file as wella s the thrid field in the foods file
-j1 2
-j2 1
foods fields
1.3
2.1
2.2
join
file #1
file #2
fields
1
2
3
1
2
counts fields
foodtypes
counts
mylist
$join -j1 2 -j2 1 -o 2.1 2..2 1.3 foodtypes counts > mylist

```
fresh      fruit       3.00
lowfat     milk        2.90
canned     milk        1.50
vegetable  soup        1.75
fresh      vegetables  7.50
```

```
milk        3
soup        8
vegetables  12
```

```
3   milk        2.90
3   milk        1.50
8   soup        1.75
12  vegetables  7.50
```

Figure 14.4. The join filter comparing different fields in each file and outputting only a few selected fields.

Join Field Delimiters

By default, the `join` filter assumes that fields are delimited with a space. If this is not the case, you need to specify the delimiter being used in your files. You do so with the `-t` option. The `-t` option takes a single character that is used as the field delimiter. The `-t` option works in the same way as the `-d` option in the `paste` and `cut` commands. For example, the option `-t:` designates the colon as the field delimiter in a file. The next example joins fields delimited by semicolons. The title fields in the **booksC** and **purchasesC** files are compared: field 1 in **booksC** (`-j1 1`) and field 4 in **purchasesC** (`-j2 4`).

```
booksC
Christmas carol:Dickens:3.50:Academic
Iliad:Homer:10.25:Random
Raven:Poe:2.50:Penguin
War and Peace:Tolstoy:15.75:Penguin
```

```
purchasesC
marylou:San Diego:CA:Christmas carol
aleina:Barrow:AL:Christmas carol
valerie:Portland:OR:Raven
larisa:San Diego:CA:War and Peace
```

```
     $ join -t: -j1 1 -j2 4 booksC purchasesC
     Christmas carol:Dickens:3.50:Academic:marylou:San
Diego:CA
     Christmas carol:Dickens:3.50:Academic:aleina;Barrow:AL
     Raven :Poe:2.50:Penguin:valerie:Portland:OR
     War and Peace:Tolstoy:15.75:Penguin:larisa:San
Diego:CA
```

You could also use a tab as a field delimiter. However, when you specify a tab as the character used for the `-t` option, you need to first quote it with a backslash. A tab on the command line is normally interpreted by the shell as just another space separating arguments. You first enter the `-t`, a backslash, and then hit the tab key. The tab character itself does not show up. In the next example, the **booksT** and **purchasesT** files use tabs to delimit each field. The user then joins the files, comparing the title fields in each: field 1 in **booksT** and field 4 in **purchasesT**. The field delimiter is specified with the option: `-t\tab`. The tab, of course does not show.

```
booksT
Christmas carol  Dickens    3.50  Academic
Iliad            Homer     10.25  Random
Raven            Poe        2.50  Penguin
War and Peace    Tolstoy   15.75  Penguin
```

```
purchasesT
marylou   San Diego   CA  Christmas carol
aleina    Barrow      AL  Christmas carol
valerie   Portland    OR  Raven
larisa    San Diego   CA  War and Peace
```

```
     $ join -t\      -j1 1 -j2 4  booksT purchasesT
     Christmas carol Dickens 3.50  Academic marylou San
Diego CA
     Christmas carol Dickens 3.50  Academic aleina  Barrow
AL
     Raven           Poe     2.50  Penguin    valerie
Portland  OR
     War and Peace   Tolstoy 15.75 Penguin    larisa  San
Diego CA
```

TABLE 14-4

The `join` filter:

`join`	The `join` filter joins the lines of different files if the values of a specified field in each file matches.
	`$ join -option file-list`
	`$ join -j1 2 -j2 1 foods counts`

`-j`*filenum fieldnum*	The `-j` option specifies what fields in each file are to be compared. Each field is numbered from 1. If you are comparing the same field in each file, you need only one `-j` option and the fieldnum. If you are comparing different fields in each file, then you need a `-j` option and *filenum* as well as the fieldnum for each file. The filenum is the position of the file's name in the file-list.
	`-j` *fieldnum* Compare the same field in each file. There is a space between the `-j` option and the *fieldnum*.
	`$ join -j 2 foods counts`
	`-j`*filenum fieldnum* Compare different fields in each file.
	`$ join -j1 2 -j2 1 foodtypes counts`

`-o`*filenum*`.`*fieldnum*	The `-o` option specifies what fields in each file are to be output. The filenum and fieldnum are separated by a period. You can list several fields, separating each filenum .fieldnum combination with a space.

	`$ join -j1 2 -j2 1 -o 1.3 1.4 2.2 2.4 foodtypes counts`
`-t`*delimiter*	The `-t` option allows you to specify your own delimiter to look for in a file. This is the same as the `-d` option in the `cut` and `paste` filters. `$ join -t: -j1 2 -j2 1 foodtypes counts`
`-a`*filenum*	The -a option outputs lines whose fields from a specified file do not match, as well as matched fields from both files.. The `-a` option takes a *filenum* to specify the file from which to output unmatched lines. `$ join -j1 2 -j2 1 -a1 foodtypes counts`

Uniq: repeated records

The `uniq` filter is designed to detect repeated lines in a file. By default, `uniq` eliminates successive repetitions of any lines from its output. In other words, `uniq` filters out any duplicates of a line. `uniq` also has options that allow you to count how many times a line is repeated, or detect and output only repeated lines. You can also look for lines that are only partially the same, comparing lines either by characters or by fields. Combined with other data filters, you can use `uniq` to examine specific fields.

`uniq` can receive input from a file specified on the command line or from the standard input. Output may be sent to a file or to the standard output. In the next example and in figure 14.5, `uniq` reads its input from the **itemlist** file and sends output to the standard output. Any duplicates of successive lines in the file are eliminated from the output. Notice that the second line in **itemlist** is a duplicate of the first, and the second line is missing in the output.

```
itemlist
chocolate milk       2.00
chocolate milk       2.00
lowfat    milk       2.00
canned    milk       2.00
fresh     vegetables 7.00
lowfat    milk       2.00
```

```
    $ uniq itemlist
    chocolate milk       2.00
    lowfat    milk       2.00
    canned    milk       2.00
    fresh     vegetables 7.00
    lowfat    milk       2.00
```

In the previous example, the third and last lines are also duplicates, and the last line is not eliminated from the output. This is because `uniq` only eliminates successive duplicate lines. To eliminate all duplicate lines, you would first have to sort the file, arranging duplicate lines next to each other. In the next example, the lines are first sorted and then piped to `uniq`. In this case both lines 2 and 6 are eliminated from the output.

```
    $ sort itemlist | uniq
    canned    milk       2.00
    chocolate milk       2.00
    fresh     vegetables 7.00
    lowfat    milk       2.00
```

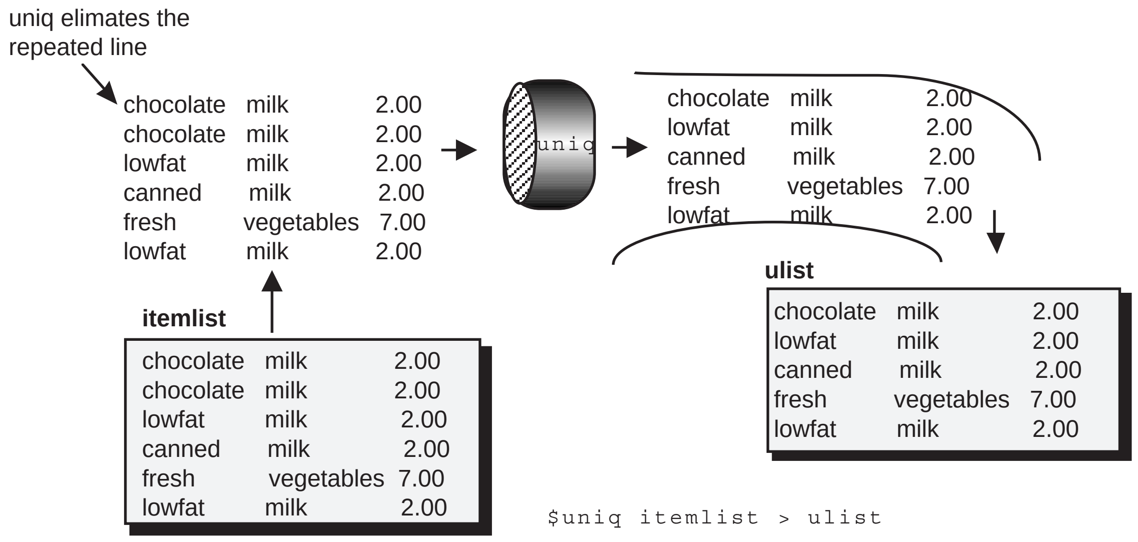


Figure 14.5. The uniq filter eliminating repeated lines.

uniq Output, Input, and Options

You can save the output of `uniq` either by redirecting the standard output or specifying a second filename as an argument on the command line. When two files are specified on the command line, the first is taken as the input file and the second is used as the output file. This differs from other filters. On the `uniq` command line, you can specify only one file for input. If you need to input more than one file you can first combine them with cat and pipe them into as standard input into the `uniq` command. In the first example the contents of **itemlist** is input to `uniq` and the output is placed in outfile. Both files are command line arguments. In the second example the contents of both itemlist and preface are combined by cat and piped as standard input to the `uniq` filter. The output of the `uniq` filter is then sent to the standard output which is redirected to the file **outfile**.

```
     $ uniq itemlist outfile
```

```
     $ cat itemlist foodlist | uniq > outfile
```

`uniq` has three standard options: c, d, and u (see Table 5). The `-c` option directs `uniq` to print out before each line the number of times the line is duplicated successively in the file. The `-d` option outputs only repeated lines. The `-u` option outputs only lines that are not repeated. In the next example, the `-c` option displays each line preceded by the number of times it has been repeated.

```
     $ uniq -c itemlist
     2 chocolate milk        2.00
     1 fresh     vegetables  7.00
     1 lowfat    milk        2.00
     1 canned    milk        2.00
     1 lowfat    milk        2.00
```

Combined with `sort`, `uniq` is able to detect duplicate lines throughout the file.

```
     $ sort itemlist | uniq -c
     1 canned    milk        2.00
     2 chocolate milk        2.00
     1 fresh     vegetables  7.00
     2 lowfat    milk        2.00
```

In the next example, the `-d` option only prints out successively repeated lines, in this case the first line.

```
     $ uniq -d itemlist
     chocolate milk          2.00
```

In the next example, the `-u` option outputs those line that are not repeated, in this case lines 3, 4, 5, and 6. Lines 1 and 2 are successively identical.

```
     $ uniq -u itemlist
     lowfat    milk        2.00
     canned    milk        2.00
     fresh     vegetables  7.00
     lowfat    milk        2.00
```

With `sort`, `uniq` can detect any duplications, not just the successive ones. In the next example, `uniq` with the `-u` option will eliminate from the output lines 3 an 6 as well as 1 and 2.

```
     $ sort itemlist | uniq -u
     canned    milk        2.00
     fresh     vegetables  7.00
```

uniq Field References

`uniq` has the capability to test for partial duplication of lines. The field option, permits `uniq` to ignore a set number of beginning fields on every line. The field option is a minus sign followed by a number, -num. The number refers to the number of fields in the beginning of a line that are to be ignored by `uniq`. Technically, a field is a set of characters delimited by a tab or a space. With the field option, a number of beginning fields on a line are ignored. Only the remaining fields on each line are tested for duplication. In the next example, the user compares only the last two fields in each line by ignoring the first field. If the last two fields in each line are the same, then they are treated as duplicate lines. The first line with the repeated fields is output. Lines 1, 2, 3, and 4 all have the same last two words: "milk 2.00". Line 5 is different: "vegetables 7.00", and line 6, though also "milk 2.0", is not successive.

```
itemlist
chocolate milk       2.00
chocolate milk       2.00
lowfat    milk       2.00
canned    milk       2.00
fresh     vegetables 7.00
lowfat    milk       2.00
```

```
     $ uniq -1 itemlist
     chocolate milk        2.00
     fresh     vegetables  7.00
     lowfat    milk        2.00
```

Using `sort` to first sort the file on the second field would allow you to eliminate all duplicates of the second and third field.

```
     $ sort +1 itemlist | uniq -1
     canned    milk        2.00
     fresh     vegetables  7.00
```

You could combine the field option with other options such as the `-c` option. In the next example, the count of each repeated line is output. Notice that the count for line 1 is 4. The last two words in lines 1, 2, 4, and 5 are the same: "milk 2.00".

```
     $ uniq -1 -c itemlist
     4 chocolate milk        2.00
     1 fresh     vegetables  7.00
     1 lowfat    milk        2.00
```

A sorted version allow you to count all duplicates in the file.

```
     $ sort +1 itemlist | uniq -1 -c
     5 canned    milk        2.00
     1 fresh     vegetables  7.00
```

Referencing fields in a file assumes that fields are delimited with specific delimiter such as a tab. However, if you have a character based file such as a fixed format file where fields are separated by padded spaces, then normal field references will not work. In this case you will need to reference character positions. The `uniq` filter allows you reference a character position and compare the remaining characters on a line. `uniq`'s character options actually ignores a set number of beginning characters on every line. The character option is a plus sign followed by a number, +num. In this case, the number refers to the number of characters at the beginning of the line that are to be ignored. In the next example the **listdataC** file has its fields separated by padded spaces rather than tabs. The +7 option instructs `uniq` to ignore the first 7 characters. After the first seven characters, lines 3 and 4 in the **listdataC** file are successively the same as well as lines 1 and 2. The first repeated line is output preceded by the number of repetitions, in this case 2.

```
itemlistC
chocolate milk       2.00
chocolate milk       2.00
lowfat    milk       2.00
canned    milk       2.00
fresh     vegetables 7.00
lowfat    milk       2.00
```

```
$ uniq +7 -d -c itemlistC
2 chocolate milk        2.00
2 lowfat    milk        2.00
1 fresh     vegetables  7.00
1 lowfat    milk        2.00
```

Using uniq with other Data filters

You can combine `uniq` with other data filters to examine different fields in a data file. In itemlist, the third word in each line is the cost of an item. In this respect, the cost of the item is the third field of a record. Using `uniq` you can find out how many items have the same price. In the next example, the first two fields are skipped in order to access the last field and a count of unique prices taken.

```
$ sort +1 itemlist | uniq -2 -c
5 canned    milk        2.00
1 fresh     vegetables  7.00
```

This does not quite give you what you need. You may really want only the value of the field you are referencing output, instead of the entire line. You can do this by using the `cut` filter to first copy out only one field in the file and have `uniq` operate on that one field. In the next example, the user first cuts the 3rd field out of each line and pipes it to `uniq` which then checks for repeated prices.

```
$ cut -f3 itemlist | sort | uniq -c
5 2.00
1 7.00
```

You could do the same thing to find out how many of each item that you have. In the next example the user cuts out the second field and pipes it to `uniq`.

```
$ cut -f2 itemlist | sort | uniq -c
5 milk
1 vegetables
```

To list the less frequently referenced items at the top of your list you could sort the output of `uniq` in sequential order. In the next example and in figure 14.6, the output of `uniq` is sorted by `sort` to insure that the less frequently used items on the list are listed first. The option `-n` for `sort` sorts in order numerically. Notice that there are two sorts. The first sorts the data so that duplicate lines will be successive. The second `sort` sorts the output of `uniq`.

```
$ cut -f2 itemlist | sort | uniq -c | sort -n
1 vegetables
5 milk
```

To list more frequently referenced items at the top of your list, you could perform a reverse sort.

```
     $ cut -f2 itemlist | sort | uniq -c | sort -nr
     5 milk
     1 vegetables
```

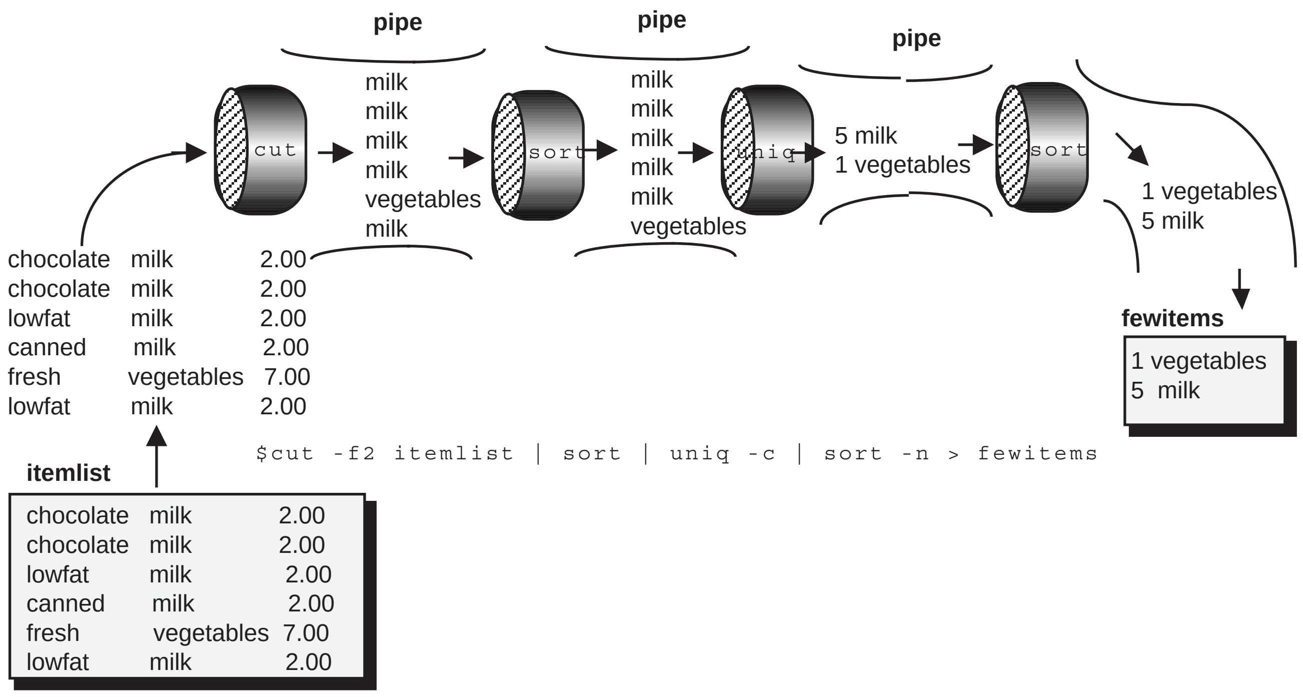


Figure 14.6. Using several data filters to form a complex query to retrieve information.

Suppose, for example, you want to find out the month in which you worked on files most. You would begin by piping the output of the `ls -l` command to a `cut` filter to cut out the month field. Then use the `sort` filter to sort the months and use `uniq -c` to count them. A sort by number would rank the output of `uniq`.

```
     $ ls -l | sed 's/  */    /g' | cut -f5 | sort | uniq -
c | sort -nr
     7 Mar
     2 Jun
     2 Feb
     1 Apr
```

TABLE 14-5

The `uniq` filter: repeated lines.

`uniq`

The `uniq` filter eliminates repeated lines from its input. You can also compare lines base on selected fields. Lines whose selected fields have the same values are considered repetitions and can be eliminated from the output

```
$ uniq options  input-file output-file

$ uniq -d itemlist newfile
```

`c`	With this option, `uniq` outputs each line preceded by the number of times the line occurs in the input. `$ uniq -c itemlist`
`d`	With this option, `uniq` only outputs repeated lines. `$ uniq -d itemlist`
`u`	With this option, `uniq` only outputs lines that are not repeated. `$ uniq -u itemlist`
-*num*	The number of fields to be skipped for comparison. Only the remaining fields are compared. `$ uniq -3 itemlist`
+*num*	The number of characters to be skipped for comparison. Only the remaining characters are compared, including spaces. `$ uniq +12 itemlist`

Database Design

Databases are often organized into several files from which you can retrieve information using a query language. In Unix, you can construct a database using several files and then use the data filters to retrieve information. Often you will need to combine data filters in complex ways, piping the output of one filter into another filter. The book database below holds information about books, publishers, and purchases. It makes use of three files: **books**, **publishers**, and **purchases**. The **books** file lists book titles, authors, price, and publisher. The **publishers** file lists publishers along with the city and state they are located. The **purchases** file list people who have bought the books and the books they purchased, as well as the city and state where they bought them. Though you cannot see them, each field is separated by a tab. This is the default delimiter for many of the data filters.

The **books** file has four data fields: the title, author, price, and publisher. The **publishers** file has three data fields: the publisher, city, and state. The **purchases** file has four data fields: the purchaser name, city, state, and title.

```
books
Tempest    Shakespeare  15.75  Penguin
Christmas  Dickens       3.50  Academic
Iliad      Homer        10.25  Random
Raven      Poe           2.50  Penguin
```

```
publisher
Penguin     Boston     MA
Academic  Cambridge  MA
Random    Chicago    IL
```

```
purchases
larisa    Sacramento  CA  Tempest
marylou   Sacramento  CA  Christmas
valerie   Portland    OR  Raven
aleina    Barrow      AL  Christmas
chris     Sacramento  CA  Iliad
justin    Napa        CA  Iliad
larisa    Sacramento  CA  Raven
```

You use key fields to retrieve information from your database. A key field is a one that you use to query a file. If you wanted to list all your book information sorted by title, you would use the title field in the **books** file as your key field. If you wanted to sort by author, you would use the author field as your key field. In the next example, the `sort` filter lists books by author. Notice how the first field is skipped and the sort is stopped at the second field.

```
     $ sort +1 -2 books
     Christmas  Dickens       3.50  Academic
     Iliad      Homer        10.25  Random
     Raven      Poe           2.50  Penguin
     Tempest    Shakespeare  15.75  Penguin
```

Certain fields are repeated in different files in order to connect the information in those file together. For example, both the **books** file and the publisher file contain a field for publisher. You can use the publisher field in each file to connect the information in books with the information in publisher. For example, you could find out the address of publishers that print a certain author. The address is in the publisher file and the author name is in the **books** file. The **books** and **purchases** file both have in common a title field. Through the title field in each you could find out who is buying what authors.

This is, of course, a very simple design. Database design employs more sophisticated strategies to connect information in different files. In relational database design there are what are called normal forms that govern the organization of key fields and database files. Due to the complexity of the subject, such concepts will not be explored here.

Using common fields to connect information in different files requires that you use the `join` filter. The `join` filter will compare the values of selected fields and, if the are the same, join and output their lines. The field value in one can match multiple instances of the same value in another file. This permits a one-to-many relationship. The one entry of a title in the **books** file can match several entries of that same title in the purchases file. There is one limitation. The files mush first be sorted on the key field. The multiple instances of a key must be sequentially combined in order to match a single instance of a key in another file. In the next example the user first sorts **books**, effectively sorting on the title field, saving the sorted version in the file **bookst**. Then the user sorts the purchases file on its title field, the fourth field in the file. The result is saved in **purchst**. Now the title fields in both **bookst** and **purchst** are sorted. The `join` filter can then join the two files based on the title fields in each.

```
     $ sort books > bookst
     $ cat bookst
     Christmas  Dickens        3.50  Academic
     Iliad      Homer         10.25  Random
     Raven      Poe            2.50  Penguin
     Tempest    Shakespeare   15.75  Penguin
```

```
     $ sort +3 purchases > purchst
     $ cat purchst
     aleina    Barrow      AL  Christmas
     marylou   Sacramento  CA  Christmas
     chris     Sacramento  CA  Iliad
     justin    Napa        CA  Iliad
     larisa    Sacramento  CA  Raven
     valerie   Portland    OR  Raven
     larisa    Sacramento  CA  Tempest
```

```
$ join -j1 4 -j2 1 purchst bookst
Christmas  aleina  Barrow     AL Dickens      3.50 Academic
Christmas  marylou Sacramento CA Dickens      3.50 Academic
Iliad      chris   Sacramento CA Homer       10.25  Random
Iliad      justin  Napa       CA Homer       10.25  Random
Raven      larisa  Sacramento CA Poe          2.50  Penguin
Raven      valerie Portland   OR Poe          2.50  Penguin
Tempest    larisa  Sacramento CA Shakespeare 15.75  Penguin
```

Instead of sorting your files each time you execute a `join` operation, you can simply create a sorted version of your file for each key field. The key field for the **books** file is the first field, the title. You could simply create a file called **booksT** that is a version of **books** sorted by title. Similarly you could create a sorted version of the purchases file using the title key field, the fourth field. In the next example the user create a sorted version for each key field. **booksT** is a sorted version of **books** based on the title, whereas **booksP** is a sorted version based on the publishers. **purchasesT** is a sorted version based on the title, and **purchasesT** is a sorted version of purchases based on the book titles.

```
     $ sort books > booksT
     $ sort +3 books > booksP
     $ sort +3 purchases > purchasesT
     $ sort publishers > publishersP
```

In the next example, the user again finds out who purchased what titles using the **bookT** and **purchasesT** files. These are the sorted versions of **books** and **purchases** based on book titles.

```
     $ join  -j1 1 -j2 4 booksT purchasesT
```

Often you will not need to display all the fields in each file. You can use the `join` filter's `-o` option to limit the fields displayed. In the next example, only the names of purchasers, the book titles, and the authors of books are listed.

```
     $ join -j1 4 -j2 1 -o 1.1 2.1 2.2 purchasesT booksT
     aleina   Christmas  Dickens
     marylou  Christmas  Dickens
     chris    Iliad      Homer
     justin   Iliad      Homer
     larisa   Raven      Poe
     valerie  Raven      Poe
     larisa   Tempest    Shakespeare
```

Using Data filters to Construct Queries

You can combine `join` with other data filters to construct complex queries. Suppose you want to know how many people in California purchased the Iliad. You can first join the **purchases** and the **books** files selecting the title and state fields. Then, using `uniq` with its `-c` option generate a list with the count of each title in a State.

```
     $ join -j1 4 -j2 1 -o 1.3 1.4  purchasesT booksT |
sort > temp
     $ cat temp
     AL  Christmas
     CA  Christmas
     CA  Iliad
     CA  Iliad
     CA  Raven
     CA  Tempest
     OR  Raven
     $ uniq -c temp | sort -t'     ' -nr
     2 CA  Iliad
     1 OR  Raven
     1 CA  Tempest
     1 CA  Raven
     1 CA  Christmas
     1 AL  Christmas
```

You can combine the entire process into one command using pipes.

```
$ join -j1 4 -j2 1 -o 1.3 1.4  purchasesT booksT | sort |
uniq -c | sort -nr
```

You can perform the same kind operations with the **publishers** file. If you want to find how many books are published in MA, you can use `join` to combine the **publishers** and **books** files and then use `uniq` to count the number of books published in each state. The key field for each file is the publisher field. The sorted version of the **books** file based on the publishers field, **booksP**, and the sorted version of the publishers file, **publisherP**, are used in the file list.

```
     $ join -j1 4 -j2 1 -o 2.3  booksP publishersP | sort >
temp
     $ cat temp
     MA
     MA
     MA
     IL
     $ uniq -c temp | sort -nr
     3 MA
     1 IL
```

You can, of course, combine this query onto one command line using pipes.

```
     $ join -j1 4 -j2 1 -o 2.3 booksP publishersP | sort |
uniq -c | sort -nr
     3 MA
     1 IL
```

The next query finds out in what cities the more expensive books are being sold.

```
$ join -j1 4 -j2 1 -o 1.2 1.3 2.3  purchasesT booksT | sort
+2nr
     Sacramento  CA  15.75
     Napa        CA  10.25
     Sacramento  CA  10.25
     Barrow      AL   3.50
     Sacramento  CA   3.50
     Portland    OR   2.50
     Sacramento  CA   2.50
```

You can even pipe the output from one `join` operation as input into another `join` operation. Remember that a minus sign, `-`, stands for the standard input when used as a file argument. You can use a `-` in place of a filename in a `join`'s file list. In the next example the output of first `join` operation is piped as the second file in the next `join` operation. The query in this example gives you a list purchasers, the authors they read, and the cities in which their books are published. First the **purchases** and **books** are combined on the title field, selecting the publisher field(`2.4`) in **books** , the purchasers field(`1.1`) in purchases, and the author(`2.2`) and in **books**. Then the resulting data is piped to a `sort` filter that sorts it on the first field, publishers. The sort output is then piped into the next `join` operation where it is combined with the publishers file on the publisher's name field, selecting the purchaser name(`1.2`) and author(`1.3`) from the combined piped data and the city(`2.2`) from the **publishers** file.

```
$ join -j1 4 -j2 1 -o 2.4 1.1 2.2 purchasesT booksT | sort
-1 | join -j1 1 -j2 1 -o 1.2 1.3 2.2  - publishersP | sort
     aleina    Dickens      Cambridge
     chris     Homer        Chicago
     justin    Homer        Chicago
     larisa    Poe          Boston
     larisa    Shakespeare  Boston
     marylou   Dickens      Cambridge
     valerie   Poe          Boston
```

Using Delimiters

Many databases may have several words to a field, including spaces between the words. For example, for a title you may need to enter "War and Peace" rather than a single word title such as "Raven". The data filters read both spaces and tabs as the default delimiters. If you want spaces to be ignored as delimiters, you need to specify a delimiter character on the command line. For `join` and `sort` the delimiter is specified with a -t option. For `cut` and `paste` it is specified with a `-d` option. If you want to use tabs as your delimiter, yet ignore spaces, you need to specify the tab as a delimiter. However, a tab as well as several other characters such as a '|', are shell metacharacters and are used by the shell for evaluate commands. To avoid such interpretation by the shell you need to quote a tab or any other shell character that you are specifying as a delimiter. You can quote a character either be preceding it with a backslash or by enclosing it in single quotes. In the case of a tab the actual tab character will not show.

In the next example, the **books** and **purchases** files have had several different records, each employing spaces in some of their fields. Each field is still separated by a tab. However, to correctly reference a field, the user need to explicitly specify that the tab is a delimiter.

```
books
War and Peace    Tolstoy  15.75  Penguin
Christmas Carol  Dickens   3.50  Academic
Iliad            Homer    10.25  Random
Raven            Poe       2.50  Penguin
```

```
publisher
Penguin   Boston     MA
Academic  Cambridge  MA
Random    Chicago    IL
```

```
purchases
larisa    San Diego   CA  War and Peace
marylou   San Diego   CA  Christmas carol
valerie   Portland    OR  Raven
aleina    Barrow      AL  Christmas carol
chris     San Diego   CA  Iliad
justin    Napa        CA  Iliad
larisa    San Diego   CA  Raven
```

In the next examples, the user again creates sorted versions of each file. However, this time, the user needs to specify the tab delimiter. The user first enter the `-t`, a backslash, and then hit the tab key. The tab character itself does not show up. The field delimiter is specified with the option: `-t\tab`.

```
     $ sort -t\      books > booksTD
     $ sort -t\      +3 books > booksPD
     $ sort -t\      +3 purchases > purchasesTD
     $ sort -t\      +3 publishers > publishersPD
```

You can, if you want, quote the tab by encasing it in single quotes. In the next example, the user queries the **booksTD** and **purchasesTD** files, specify the delimiter with a quoted tab: `-t'tab'`. The tab, of course, does not show up.

```
     $ join -t'     ' -j1 4 -j2 1 purchasesTD booksTD
Christmas carol  aleina  Barrow    AL Dickens 3.50 Academic
Christmas carol  marylou San Diego CA Dickens 3.50 Academic
Iliad            chris   San Diego CA Homer   10.25 Random
Iliad            justin  Napa      CA Homer   10.25 Random
Raven            larisa  San Diego CA Poe      2.50 Penguin
Raven            valerie Portland  OR Poe      2.50 Penguin
War and Peace    larisa  San Diego CA Tolstoy 15.75 Penguin
```

Chapter Summary: Data filters

The `sort`, `cut`, `paste`, `join`, and `uniq` filters perform data operations on an input data stream. Input is received either from files or the standard input. The filters then outputs selected lines from the input. This output can then be directed to a file or a device like a printer. The original input is not touched. If a file is used as input, it remains unchanged.

The `sort`, `cut`, `paste`, `join`, and `uniq` filters perform data operations on files that have been organized into fields and records. The `sort` filter sorts records according to a specified field. The `cut` filter outputs only specified fields in a record. The `paste` command combines the records in data files. The `join` filter selectively combines records in data files. The `uniq` filter deletes duplicate lines.