GroupBy: A Grouping Terminal Filter

Dylan Laufenberg (lauf@pdx.edu)
Portland State University

Introduction

This paper introduces the concept of a grouping terminal filter and a reference implementation called groupby[1] written in Rust. A **grouping terminal filter** accepts input via standard input, groups the input in some useful way, and passes each group of input to any subsequent filters.

Terminal filters

On Unix-like systems, when the user runs a terminal program, the shell provides the program's process with a standard complement of three input and output streams that it can use to read text from the user or write text to the screen. **Standard input (stdin)** is an input stream that receives any text the user types. **Standard output (stdout)** is an output stream that asks the shell to write text to the screen. **Standard error (stderr)** is an output stream, like stdout, meant for error text. Conceptually, all three of these are unidirectional pipes: text flows through each pipe from a single sender to a single receiver.

This paradigm is powerful in part because most shells allow the user to remap these three pipes when running a program. For instance:

```
wc < input.txt \
   > output.log \
   2> error.log
```

invokes the wc program, overriding all three standard streams. The shell reads the file input.txt and provides its content to wc as if the user had typed it. If and when wc writes to stdout or stderr, instead of appearing on-screen, these writes go to the files output.log and error.log, respectively. This paradigm also lets users compose command chains, connecting one program's stdout to another program's stdin. For instance, writing:

```
ls | wc -l
```

connects the ls process's stdout to the wc process's stdin, counting the files in the current directory.

The flexibility of this system gives rise to composable programs called **terminal filters** that are designed to be chained together to solve common problems.

A grouping filter

For some tasks, it's necessary to break input into groups and perform a task on each group. For example, when processing lines of input that include dates, one might wish to group lines by date. Given a directory hierarchy containing text files, one might write:

```
find . -print0 | xargs -0 wc
```

to count the lines, words, and characters in all files. If the filenames include dates, one might notionally wish to write something like:

```
find . | \
groupby --year --print0 | \
xargs -0 wc
```

to group the files by year and count the lines, words, and characters in each year's files. Unfortunately, it's impossible to write the exact filter described in this chain.

The problem

In processing this filter chain, the shell connects the groupby process's stdout to the xargs process's stdin. This is essentially invisible to both processes. There is no particularly good way for a filter to inspect the other filters in a command chain. Thus, groupby has just one instance of xargs and no way to create more. For groupby to work correctly, it must invoke the rest of the command chain once for every group, and the standard I/O system provides no way to accomplish this task. By rewriting the previous example to pass groupby the text of the xargs command as an argument, it becomes possible to write a groupby filter that will invoke the command once per group:

```
find . | \
groupby --year \
        --print0 \
        -c "xargs -0 wc"
```

Defining groups

Having resolved the technical hurdle of invoking a command chain once per group, the primary remaining issue is how to define groups. Equivalence classes provide a natural answer. In mathematics, an **equivalence**

[1] https://github.com/edev/groupby

class is a group of elements within a set that are equivalent in some way. For instance, given a set of integers, one might divide the set into even and odd equivalence classes or negative and nonnegative equivalence classes.

In processing a set of lines of text, types of equivalence abound. For instance, one might define equivalence as equality on the first *n* characters, the last *n* characters, and so on. Perhaps the most generally useful definition, however, is equivalence on regular expression matches. To match the first 2 characters of a line, for instance, the regular expression `^.{2}` will suffice. Given the lines:

```
Alphabetical
Brave
Bravo
Cavalry
```

the above regular expression will divide the lines into three equivalence classes:

"Al"	**"Br"**	**"Ca"**
Alphabetical	Brave Bravo	Cavalry

Although many solutions exist, regular expressions are a widely useful and widely known tool for text processing. This makes regular expressions a natural way to define equivalence classes over lines of text.

GroupBy

The reference implementation of a grouping terminal filter, named GroupBy, defines groups by equivalence on regular expression matches as described above. While this does not cover every possible way to group lines of text, it assuredly covers an enormous swath of common cases.

GroupBy utilizes regular expressions to easily group text files by year and count the lines, words, and characters in each year's body of text. The real-world data set that motivated the development of GroupBy is an archive of text files from 2012 on, all with .txt extensions. They sit alongside some non-text files that must be excluded. The full, working command to count the lines, words, and characters of each year's body of work is as follows:

```
find . -iname '*.txt' | \
groupby -r "\d{4}" \
        --print0 \
        -c "xargs -0 wc | tail -n 1"
```

Running this command yields tallies for each year:[2]

```
:
  98 1290 7419 total

2012:
   860  22369 119402 total

2013:
  3541  97003 526686 total

2014:
  2805  82319 445562 total

2015:
   999  36401 199237 total

2016:
   7281  194784 1056327 total

2017:
  23916  694027 3678968 total

2018:
  24341  704840 3749280 total

2019:
  24358  614774 3290123 total

2020:
  5026 125309 674516 total
```

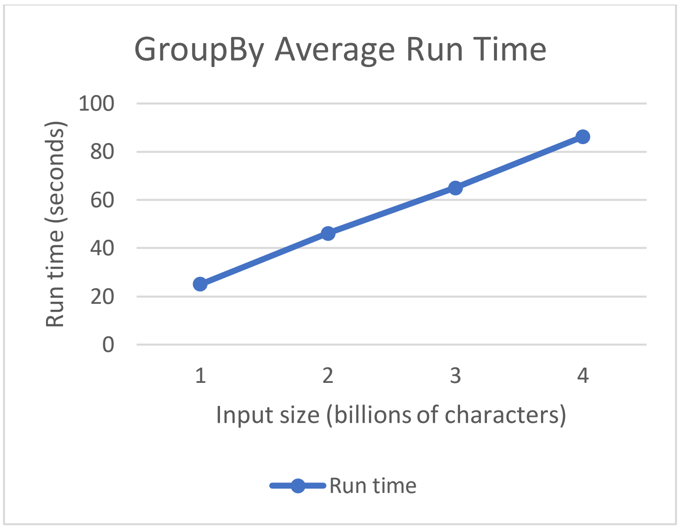
Performance

A brief benchmark series presented here demonstrates the performance scaling of GroupBy as the size of input grows. First, four files were generated, with 1, 2, 3, and 4 billion characters randomly selected from uppercase and lowercase English letters and the newline character. The program was benchmarked with each of these files as input, outputting to /dev/null to best isolate the time required by GroupBy (rather than printing to the screen). The following graph and table present the average of 5 runs at each input size of the following command:

```
groupby -f 4 < file > /dev/null
```

Input size (characters)	**Average time (seconds)**
1 billion	24.97
2 billion	46.20
3 billion	65.00
4 billion	86.24

[2] The blank category counts the few files without dates.



In this benchmark, GroupBy's run time scales linearly with input size.[3]

Correctness

The Rust library that underlies GroupBy, providing its key data structures and equivalence grouping, is unit-tested for correctness. Due to the I/O-heavy nature of the application that wraps the library, the application code is manually tested and carefully verified.

Related work

The POSIX standard find utility[4] is indispensable for searching directory hierarchies. xargs[5] translates standard input to command-line arguments, blurring the definition of a filter in similar ways as GroupBy. To the best of the author's knowledge, there currently are no grouping terminal filter programs in the POSIX ecosystem.

Conclusion

The grouping terminal filter fills a void in the POSIX ecosystem. It suits a narrow set of use cases that are unserved by the current set of terminal tools. Most users and most tasks will not call for its use. When needed, however, it is an invaluable tool to automate grouping tasks that would otherwise need to be performed by hand.

[3] Benchmark system specifications: Ryzen 7 3800X, 2x8GB DDR4-3866, 1933 MHz fabric and memory clocks, PCIe 3.0 NVMe SSD.

[4] https://en.wikipedia.org/wiki/Find_(Unix)

[5] https://en.wikipedia.org/wiki/Xargs