Efficient Vocabulary Discovery in Late Antique Texts at the UNIX Command Line

Andrew J. Hayes

May 22, 2025

Introduction

This presentation concerns a workflow for discovering vocabulary in digitized late antique texts. I present this workflow first by identifying the use cases for it and outlining the goals this presentation hopes to achieve and those it does not. A conceptual overview follows, with the aim of providing just enough background knowledge to use the workflow. I present the workflow in overview and interactively demonstrate its core steps. I conclude by highlighting the workflow's limitations and imparting further resources that will enable interested attendees to experiment with the workflow themselves.

The examples concern the author's area of expertise: which is the corpus of the fourth century poet and theologian, Ephrem the Syrian. Most of Ephrem's works were critically edited in the previous century by Edmund Beck, for which he also provided high quality German translations.[1] The demonstrations in this talk will use these German translations. But the methods presented here can apply to any digitized text in LTR unicode script.

Use Cases

The workflow demonstrated here is designed to address a particular difficulty in patristic scholarship: searching ancient text(s) (whether in the original or in translation) for which no born digital search tool is available or accessible. Many collections of primary sources have such tools. The Digital Syriac Corpus is an example of a native digital collection with robust search tools. The Thesaurus Linguae Graecae is another. Using such tools, if available, is ordinarily preferrable to the approach described in this presentation.

[1] Sebastian Brock, "Bibliographical Handouts by Brock" (https://syri.ac/brock, 2025). See in particular "St. Ephrem: A Brief Guide to the Main Editions and Translations."

But what to do if no such tool exists, is pay-walled, or is inaccessible for some other reason? Many scholars maintain their own corpora of scanned pdfs for private research use. Optical character recognition (OCR) makes digital search of such texts possible. This presentation shows how to search a local directory tree of OCR'ed pdf files of primary sources in original or in translation in an efficient manner using freely available tools in the UNIX programming environment. These methods are imperfect, but nevertheless highly useful for assembling a working vocabulary list as a starting point for careful reading and research. Accompanying this presentation is a public repository on github with sample command-line recipes and instructions allowing any scholar to download and experiment with them.

Note that even when a native digital search tool is available, it is sometimes convenient and practical to employ the tools and techniques presented here. The following are four such cases:

1. One wants to search a large number of pdf files from disparate corpora at once.
2. One wants to search in multiple languages simultaneously.
3. One wants to automate or script complex queries for reproducibility and convenience.
4. One wants to produce a customized report of search results for use in another application.

The examples in this talk involve searching Ephrem the Syrian's corpus, mostly in the form of Edmund Beck's German translations, using scans made for personal use from purchased physical volumes. The tools and techniques are, however, applicable to any pdf containing LTR text in a Roman or Greek alphabet, or any such script supported by a terminal emulator.

Three Core Steps (Goals and Non-Goals)

These workflows are complex. It is impossible to describe all their parts fully within the scope of this presentation. Some of the parts omitted from the main discussion are identified and augmented in the Github repository with hints and suggestions for how one might best accomplish the task. Nevertheless, the main business of this talk is to discuss the three core steps in the workflow:

1. Repaginate
2. Explore
3. Report

Background

UNIX and the Command Line

UNIX is a family of operating systems descended from an operating system developed at Bell Labs in the 1970s.[2] Its original headline feature was the ability to provide robust multi-user support in a way that clearly delineated user ownership of files to prevent conflicts.[3] It eventually became a kind of standard: POSIX, the Portable Operating System Interface.[4] Today, MacOS and various forms of Linux and BSD are the most commonly used UNIXes. They form the backbone of the internet. Most servers run some form of UNIX.

UNIX OS's share a common structural feature: they consist of a kernel and a shell. A shell is a textual interface for users to the kernel of the OS. The shell is a command interpreter. It provides a prompt and a command language allowing the user to issue written commands to the kernel and to orchestrate the operation of many programs simultaneously. Although there are many different shells available to a user, most implement some substantial portion of the POSIX standard, which means that a user who learns one shell can usually apply that knowledge to any UNIX installation. The tools discussed in this presentation are as standard as possible and should be available for any UNIX system.

On modern desktop and laptop computers with a window-based graphical user interface, the user accesses the shell through a terminal emulator program, which provides a place in which to issue commands via the shell and to receive output from those commands. Commands are issued to the shell as lines of plain text. This constitutes the Command Line Interface (CLI). We will be extracting information from pdf files

[2] Brian W. Kernighan and Rob Pike, *The UNIX Programming Environment* (Englewood Cliffs, New Jersey: Prentice-Hall, 1984), vii.

[3] Kernighan and Pike, *The UNIX Programming Environment*, 1.

[4] Arnold Robbins and Nelson H. F. Beebe, *Classic Shell Scripting: Hidden Commands That Unlock the Power of Unix* (Sebastapol, California: "O'Reilly Media, Inc." 2005), 1–7.

and manipulating it at the command line using the tools `pdfgrep`, `grep`, `awk`, and `lualatex`.[5] Some information about how to install `pdfgrep` via common package managers and `lualatex` as part of a TEXdistribution is provided in the acommpanying repository. The tools `grep` and `awk` are already included in any POSIX compliant environment. Any standard shell may be used. My examples will use `zsh`, the default in MacOS.

One important feature of UNIX tools is their composability. They can be chained together into a pipeline to achieve a series of transformations producing the desired result. Textual information flows through UNIX commands like water flows through a pipe.

Key Concepts: PDF Page Labels and Regular Expressions

The heart of this workflow turns on two key concepts: pdf page labels and regular expressions. Such regexes, as they are called will not receive their own tutorial here, but simple examples in the demonstration portion will illustrate how they are used. Resources for further study are available in the associated github repository. In essence, a regex is a plain text string that represents a pattern. A regex engine evaluates that string and finds all the strings in the source document that match the pattern. As originally conceived, regexes are used by a tool such as `grep` to search one or more plain text files. The tool `pdfgrep` is a free and open-source variant of `grep` that makes it possible to search a pdf file. Like `grep` the `pdfgrep` tool is given a regex and one or more source files and outputs all the matches, along with useful context for the match: for instance, the filename of the source file in which the match occurs and the pdf page label of the page on which the match occurs.

Page labels are a form of metadata in a pdf file that are displayed by most pdf viewing software, such as MacOS `Preview.app` or KDE `Okular`. They usually indicate a digital text's logical page number corresponding to its printed original. As in a printed text, cover pages, frontmatter, body, and backmatter can have distinct pagination. Thus, frontmatter might be paginated with lowercase roman numerals, while the body might have arabic numerals. As a result, a given page might, in absolute terms, be the seventh page in a pdf document, but be labeled with the number 2 because

[5] Other forms of LATEX, such as `pdflatex` and `xelatex` will serve just as well.

it has been preceded by a cover page and pages i-iv of frontmatter. The PDF Association describes these labels as “an optional descriptive label of a page that is commonly presented on-screen. This is in contrast to the integer page index used internally in PDF files.”[6] Such labels are useful for working with a digital version in concert with its printed *Vorlage*.

Correspondence to the printed original is key to this workflow. If the labels of the scanned pdf correspond correctly to the printed original, the list of matches produced by **`pdfgrep`** can easily be looked up in either the digital or printed version. Moreover, PDF viewer software typically provides a keyboard shortcut to jump directly to a specified page label, a feature that is important when scanned files run to hundreds of pages and lack other forms of navigable structure.

Unfortunately, a dumb scan of the printed original needs the page labels added, and most free viewers provide limited functionality, or none at all, for editing page label metadata or page order. Hence the first step in the workflow is to repaginate using LaTeX. We turn now to the workflow.

Workflow

Cautionary Note: When working with any important file, such as laboriously obtained scans of physical originals, it is good practice always to edit copies rather than overwriting the original files.

Pre-requisites

I take for granted that you already have one or more ocr’ed pdf files that meet the following requirements:

1. scanned at 300 dpi or better;
2. OCR’ed using a high quality OCR engine (Abbyy recommended);
3. and single page: each page of the hard-copy corresponds to a single page in the pdf.

Phase 1: Repaginate

We will use the accompanying file `repaginate.tex`. We start with the scanned pdfs that need repagination in a dedicated `sources` directory. We

[6] PDF Association, “Glossary of PDF Terms” (https://pdfa.org/glossary-of-pdf-terms/#p, 2025).

then make any necessary changes to the `repaginate.tex` file and typeset it.

In our example, the original file is Beck's translation of the *Hymns on the Nativity* with the filename `hdn-ocr-optimized.pdf`. We first examine the scan to determine the absolute page numbers of each section requiring distinct pagination:

- 1-14 should be numbered i-xiv
- 15-end should be numbered 15-226

In `repaginate.tex` we comment out lines 38-39 because we don't need any cover pages. Then, we change line 54 to include pages 1-14, and the filename to `sources/hdn-ocr-optimized.pdf`. We change line 57 to include the rest of the pages 15-end using the string `15-`. Once again we must update the filename to `sources/hdn-ocr-optimized`. This will compile a new pdf with the specified pages, using the specified page labels.

The final compilation results from the following command:

(1) **Repagination via lualatex**

```
lualatex -output-dir=build repaginate.tex
```

The result should be a pdf file with the correct page labels.(For the purposes the demonstration, I will move the file to the `corpus` directory: `mv repaginate.pdf ./corpus/HdN\ German.pdf`)

At this point the file is ready to use for searches. One can build up several such pdfs in the same way, place them in a single directory, and search them all at once. For this demonstration we are also going to establish a file naming convention that will help produce a useful report at the end. The convention is convenient, but arbitrary, and is necessary only if you want to use my `awk` script without modification. You are free to re-write it to follow some other convention. The included `awk` script is designed to use pdfs with filenames that begin with an abbreviation designating the collection, followed by a space, followed by any other text. For example, if one has the *Hymns on the Church*, the *Hymns on Faith*, and the *Metrical Discourses on Faith*, the filenames for each would be: `HdE <whatever>.pdf` and `HdF <whatever>.pdf` and `SdF <whatever>.pdf`.

Phase 2: Explore

Following the UNIX philosophy, we first compose small searches interactively at the command line. Once one has worked out the pieces, they can be put together into a single pipeline.

(2) **Getting a list of matches from a single text**

This basic search shows how regular expressions can be useful for capturing text with and without diacriticals, and for capturing compounds:

```
pdfgrep -e '[Ss]ch[aä]tz' HdE\ German.pdf -H --page-number=label
```

Using the `-H` option forces the filename to be output when there is only a single text being searched.

(3) **Counting the number of matches**

```
pdfgrep -e '[Ss]ch[aä]tz' HdE\ German.pdf -H -c
```

Character classes inside the square brackets will match any of those characters.

(4) **Quickly eyeballing the number of matches across different texts**

```
 pdfgrep -e '[Ss]ch[aä]tz' HdE\ German.pdf HdV\ German.pdf -c
```

The `-c` flag is used to count the number of matches, per file. The result shows that although the HdV and the HdE are comparable in line count, HdE seems to use the language of treasure more frequently.

(5) **Dealing with lower quality OCR and spelling variation**

```
 pdfgrep *.pdf -e 'G[aei]hen(n)?a' --page-number=label -H
```

Here we search for the term "Gehenna" all the pdfs in the current directory. The expression `*.pdf` is an example of a glob. The shell expands the star character to a string of any length, which means that our search will match all the filenames in the directory that end with the extension "pdf." The proper name Gehenna admits of more than one spelling, sometimes due to poor quality OCR. But the same feature is also useful for searching in multiple languages at once.

(6) Searching in multiple languages simultaneously

Another way to do that appears in the following code snippet:

```
pdfgrep -i -e '([Kk]ingdom)|(Königtum)|(\(Himmel\)[Rr]eich)' --page-number=label -H *.pdf
```

The pipe character in a regular expression serves as a logical `OR`. This permits us to search for the concept of a kingdom in multiple languages. If we had, for instance, Leloir's French translation of Ephrem's *Diatessaron* commentary, we could add the French word for kingdom to the regex, using the pipe character. To make sure that the logical `OR` separates whole words and not single characters, we group them using round parentheses.

(7) Lookbehinds and pipelines when dealing with many false positives

```
pdfgrep  -P '(?<![Zz]u )Ende(?! des [a-z])'
--page-number=label -H -C 3
--color=always *.pdf | grep -v -e '[Zz]u'
```

In this example, we find instances of the word *Ende* but which are not preceded by the word *Zu*, because the phrase *Zu Ende* is very common. It is Beck's usual translation of *šlem*, a scribal note in the mss. typical at the end of a *madrāšâ* or collection of *madrāšê*. Being a scribal note, it is of no interest for understanding Ephrem's word usage, so we exclude it in the present inquiry.

Doing this requires the use of a negative lookbehind, which is not a feature of basic or extended regular expressions.[7] Thus we must invoke `pdfgrep` with perl compatible regular expressions, a more sophisticated tool. We invoke it via the `-P` flag. The negative lookbehind itself is `(?<! . . .)` and the negative lookahead is `(?! . . .)`. This excludes matches in which the word *Ende* is preceded or followed by the most common patterns we wish to exclude.

But there's still a problem. OCR'ed texts often have inconsistent whitespace between words, and thus there are still some passages in which the undesired matches are included due to variable whitespace. Lookbehinds and Lookaheads require fixed length strings, so we cannot account for the variability that way. Thus we pipe the results into a second invocation

[7] "Advanced Grep Topics" (https://caspar.bgsu.edu/~courses/Stats/Labs/Handouts/grepadvanced.htm, 2023); Michael Fitzgerald, *Introducing Regular Expressions* (Sebastapol, California: "O'Reilly Media, Inc." 2012), 78.

of `grep` to filter out any instances of *Zu* remaining. The inversion flag `-v` accomplishes this nicely. The result is not perfect, but it excludes a sufficient number of false positives that any remaining ones can be discovered manually.

Here's a version that omits the `-C` flag, which is not useful when using the command in a pipeline to produce a report.

(8) **Modified Version**

```
pdfgrep  -P '(?<![Zz]u )Ende(?! des [a-z])'
--page-number=label -H
--color=always *.pdf | grep -v -e '[Zz]u'
```

Phase 3: Report

By default, these commands simply print the information they produce to your screen, referred to as standard output: `stdout`. Once you've built up a number of small searches that can easily be recalled using your shell's history function, you may wish to save the results to a file so that you can use them in the future. The easiest way to do this is to simple redirect to output stream to a file, using a redirection operator: `>`.[8] For example:

(9) **Redirection example**

```
pdfgrep -e '[Ss]ch[aä]tz' HdE\ German.pdf
-H --page-number=label > results.txt
```

The result is a plain text file that matches what is output on screen, but without any colors. It is possible to output this information into even more useful formats with the help of another standard UNIX tool: `awk`. Like `grep` `awk` uses regexes. Its data model is to loop through all the lines of a file, performing tasks depending on what it finds. In this case, we can use it to transform the output of pdfgrep into a csv (comma separated value) file that can be opened in a spreadsheet application or fed into a data analysis pipeline.

The repository includes a very simple awk script (`grep_to_csv.awk`) that parses the output of `pdfgrep` and structures it as a csv file:

[8] Chet Ramey and Brian Fox, *The GNU Bash Reference Manual*, 5.2 ed., 2022, sec. 3.6.

```
BEGIN { FS = ":"; print "text", ",", "citation" }
{
    cutoff = index($1, " ");
    # print "cutoff index is " cutoff # for debugging purposes
    print substr($1, 1, cutoff) ",", $2
}
```

This awk script begins by setting the field separator to `:`, which always appears when `grep` is invoked with the `-H` flag. Doing this causes the first field or chunk of text on each line to be the filename in which a given match was found. It then uses `awk`'s built in string `index()` function to determine where a space occurs in that first field first field and the `substr()` (substring) function to extract the first part of the filename before the space.[9] Per our naming convention, this substring constitutes the abbreviation of the work in question: *HdE* or *SdF*, for example. And the `print` instruction prints that abbreviation, followed by a comma, followed by the page number produced by `pdfgrep` for each match. The result is a simple csv file.

Here is an example of the whole pipeline in action:

(10) **Report pipeline**

```
 pdfgrep *.pdf -e 'G[aei]hen(n)?a'
  --page-number=label  | awk -f grep_to_csv.awk > gehenna.csv
```

This produces a file called `gehenna.csv` containing all the instances where the term "Gehenna" appears in all pdf files in the directory.

Summary and Limitations

UNIX command line tools, when put together, constitute a powerful, flexible, and free way to construct a searchable corpus of pdfs from physical originals, explore the corpus, and produce usable reports to show approximate information about the location, frequency, and distribution of vocabulary of interest.

Because the OCR on which it relies is imperfect, this workflow requires

[9] Alfred V. Aho, Brian W. Kernighan, and Peter J. Weinberger, *The AWK Programming Language* (Addison-Wesley, 1988), 41–43. This is the original manual, which remains relevant. `Awk` is included in the POSIX standard, but the most up to date version can be found at onetrueawk.

care and attention to use in a responsible way. It will produce false positives. It will miss some (possibly) important passages in which the vocabulary of interest appears. Matches should always be humanly verified. And of course, a list of matches functions best as a guide for reading the original sources, which is the goal of this workflow. It points out the likely places for further study, much as a print or digital concordance does. The difference between this approach and a concordance is that you are able to construct the concordance to suit your exact requirements, and that you can update it over time as you improve and curate your corpus of scanned texts.

Bibliography

"Advanced Grep Topics." https://caspar.bgsu.edu/~courses/Stats/Labs/Handouts/grepadvanced.htm, 2023.

Aho, Alfred V., Brian W. Kernighan, and Peter J. Weinberger. *The AWK Programming Language*. Addison-Wesley, 1988.

Brock, Sebastian. "Bibliographical Handouts by Brock." https://syri.ac/brock, 2025.

Fitzgerald, Michael. *Introducing Regular Expressions*. Sebastapol, California: "O'Reilly Media, Inc." 2012.

Kernighan, Brian W., and Rob Pike. *The UNIX Programming Environment*. Englewood Cliffs, New Jersey: Prentice-Hall, 1984.

PDF Association. "Glossary of PDF Terms." https://pdfa.org/glossary-of-pdf-terms/#p, 2025.

Ramey, Chet, and Brian Fox. *The GNU Bash Reference Manual*. 5.2 ed., 2022.

Robbins, Arnold, and Nelson H. F. Beebe. *Classic Shell Scripting: Hidden Commands That Unlock the Power of Unix*. Sebastapol, California: "O'Reilly Media, Inc." 2005.