Python vs. AWK, Perl, Julia: Processing large data files

Pierre S. Caboche

July 2, 2022

Abstract

In this article we will learn how to perform a very useful task (processing a large text file, splitting data into “buckets”) using three different scripting languages: Python, AWK, and Perl. This will allow use to compare those four languages in terms of features, syntax, and performance.

This will also give us the opportunity to learn more about these languages. Our examples and explanations should help you easily get started, even if you have no prior experience with Python, AWK, Perl, or Julia.

Target audience

This article might be of interest for:

- programmers in general
- people who need to quickly get started with Python, AWK, Perl, or Julia
- Data Engineers, who regularly need to write scripts for processing large files
  - in our example, we will split data into “buckets”.
  - specifically, we will take a MariaDB database dump file, and separate the data on a per-table basis

Introduction

Python is a general-purpose language which has become extremely popular in domains such as Big Data, Data Engineering, Machine Learning, etc. Python also has a number of applications in other fields due to its flexibility and relative ease of use.

In this article, we will try to use Python to process a large text file (a very common problem in Big Data and other fields), evaluate Python's abilities to accomplish this task, and then compare it with solutions using other scripting languages (namely: AWK, Perl, and Julia).

The problem we are trying to solve is the following: we have been given a huge text file, and its data needs to be separated into several smaller, more manageable files ("buckets" of data) based on certain conditions.

In our example, the file will be a "backup" (dump file) of a MySQL / MariaDB database, as produced by the `mysqldump` utility, and we need to separate the data on a per-table basis (1 file per table).

I have chosen this example because:

- it should be easy to follow (we will deliberately keep the examples simple)
- it touches different domains
  - Data Engineering (for the main concept)
  - MySQL / MariaDB database administration (for the example)

Scripting languages

To process our files, we will be using scripting languages only (Python, AWK, Perl). The advantage of scripting languages is that the scripts can be easily edited, without the need to recompile the code into an executable.

Our scripts will be compatible with the Linux Command Line Interface (CLI). On Windows, it is possible to run such scripts using Windows Subsystem for Linux (WSL) 2.

In this article, we see how to implement a very common problem (processing a text file), in the following scripting languages:

- Python
- AWK
- Perl
- Julia

For each language, we will see how to perform some common operations. This will not only allow us to easily get started with these languages, but it will also allow us to compare how these operations are performed in each language:

- General
  - how to run the script
  - variables (strings, integers)
  - string operation (concatenation, formatting)
- Files and I/O
  - how to read a stream of data from the standard input (`stdin`)
  - how to open and close files
  - how to write to standard output (`stdout`), standard error (`stderr`), and files
- Regular expressions
  - how to test if the data matches a certain *regular expression*
  - how to capture groups in a *regular expression* (named groups, vs. numbered groups)

The general problem

The type of problem we're trying to solve is of the following form:

- we have a text file containing a lot of data
- this data needs to be split into several smaller "buckets" (one file per bucket)
- we are reading the input file line by line, and writing to different output files
- by analysing the content of a line, we decide when to switch between the different output files

For example, we may have a file with the following template:

```
# The following needs to go to file #1
Some data here...
(...)

# The following needs to go to file #2
More data there...
(...)

# The following needs to go to file #1
Even more data, which needs to go to the first bucket...
(...)
```

In this example, the "bucket switch" is indicated by a line of the form:

```
# The following needs to go to file ...
```

...and we determine which bucket to switch to based on the content of the line.

General Information

This document was first published at:

https://pcaboche.github.io

Legal

Last revision: 2 July 2022

- Singular terms shall include the plural forms and vice versa.
- this Document is Copyright 2022 Pierre Caboche. All rights reserved. No unauthorised distribution allowed.
- the LaTeX Code which produced this Document is Copyright 2022 Pierre Caboche. All rights reserved.
- this Document also features Source Code (also known as Code Snippet), subject to different licenses.
  - if no license is mentionned, it is to be assumed that the Source Code is *proprietary*, and the property of its author and copyright holder.
  - if a license is mentionned (in a comment or through other means), you need to refer to the full description of the license.

    Below is an example of comment, indicating that the featured Source Code is subject to the BSD License, with mention of the copyright year and copyright holder:

    ```
    # This source code is under the BSD License
    # Copyright 2022 Pierre S. Caboche
    ```

- all of the Content (including Document, LaTeX Code, Source Code) IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS ”AS IS” AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL THE COPYRIGHT HOLDER OR CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.
- the Copyright Holders reserve the right to amend this Legal Notice at any time, without prior notification. The latest version of the Legal Notice supersedes and replaces all prior versions of it.

BSD license

Wherever the BSD license is mentionned, the following applies:

Contents

Part I

MySQL / MariaDB : splitting a database dump file, on a per-table basis

This specific problem will serve as our example throughout the article:

- we’ve been given a big `.sql` file, which is the result of running `mysqldump`to “backup” a MySQL/MariaDB database
- the `.sql` file contains the data from all our tables
- we need to split that data into several `.sql` files, one file per table
- the string `-- Table structure for table `table_name`` marks the beginning of the definition for table `table_name`, followed by its data.
- Whenever we encounter such a string, we need to change the output of our script (to a file named `table_name.sql`)

In this problem, each table constitutes a “bucket”.

If, for example, we wanted to separate the table structure from its actual data, this would constitute 2 buckets per table (one for the structure, and one for the data). To make our script easier to understand, we will keep 1 bucket per table.

What’s important to remember is:

- a “bucket” represents how the data is separated *logically*
- a file is how the bucket data is stored *physically*
- we decide to make just 1 bucket per table (and therefore, 1 output file per table)

Note on the example

Our database “dump” file actually contains more than the tables’ definition and data. For example, it may contain definitions for views, stored procedures, and other definitions.

A fully working example would require more complex logic, and may also depend on external factors which are out of my control (such as: how the ”dump” file was generated in the first place, the version of `mysqldump`, and others).
We have deliberately kept the examples in this article simple and easy to follow.

The scripts provided in this article are FOR ILLUSTRATION PURPOSE ONLY, USE AT YOUR OWN RISK.

What the scripts will do

Our scripts will:

- read data from the standard input (`stdin`)
- write to different files based on some conditions

The output file names will follow the same form:
`${number}_${table_name}.sql`

For example: `0041_some_table_name.sql`

We are numbering our files in order to reassemble them later (and in the correct order), and compare the input with the output of our scripts (this will be important later…)

The output files will be stored in an output directory. We will use a different output directory for each of our implementations (`python/`, `awk/`, `perl/`). This will be useful later, to compare the output results.

Tools

Here is a typical example of how we will run a script:

```
$ time pv input_file | ./script.py
```

We use the linux `time` command to measure how long a script took to execute.

We use *pipes* (|) to pass data from one process to another. The output of one process (`stdout`) is passed to the standard input (`stdin`) of the next.

Our scripts will read data from the standard input (`stdin`). This approach is generally more flexible that reading data from files.

We prefer to use the `pv` (*pipe viewer*) command instead of `cat` (or the < file indirection).

`pv` will not only read the file, but also provide extra information, such as: the file size, *current* read speed (in a unit such as MiB/s), and overall progress. This immediately tells us when the read/processing speed is abnormally slow.

Please note however, that when talking about the *overall* read speed, we divide the file size by the real time (as reported by `time`), not what is displayed by `pv` (which is the *current* read speed at the end of the file processing).

Execution environment

I first ran the experiment on a computer equipped with a fast SSD on NVMe PCIe gen 3. We'll call this computer *"gen-8"*, for it has an 8th generation Intel® Core™ i5 processor.

Later on I got access to a more recent computer (11th generation Intel® Core™ i5) with a faster SSD (PCIe gen 4). I re-ran the tests, and included them for comparison. We'll call this computer *"gen-11"*.

Below is some information about the hardware and software used...

Table 1: Hardware comparison

	Computer 1	Computer 2
Name	gen-8	gen-11
CPU	Intel® Core™ i5-8259U CPU 8 × 2.30GHz (max: 3.80 GHz) Released: Q2'2018	Intel® Core™ i5-1135G7 8 × 2.40GHz (max: 4.20 GHz) Released: Q3'2020
RAM	16 GB (2 × 8GB, DDR4-2400 MHz)	16 GB (2 × 8GB, DDR4-3200 MHz)
SSD	NVMe PCIe *gen 3* Seq. read: $\sim$2.3 GB/s Seq. write: $\sim$1.7 GB/s Partition format: XFS	NVMe PCIe *gen 4* Seq. read: $\sim$6.5 GB/s Seq. write: $\sim$4 GB/s Partition format: XFS

Table 2: Software versions

OS	Linux (Fedora 35)
Python	3.10.5
GNU Awk	GNU Awk 5.1.0, API: 3.0 (GNU MPFR 4.1.0-p13, GNU MP 6.2.0)
Perl	v5.34.1
Julia	1.7.3

I also tested the scripts on a Windows 10 laptop, using WSL 2 (Windows Subsystem for Linux, version 2).

The scripts run well under WSL 2, but that laptop had a slower SSD, which impacted the results (some scripts were bottlenecked by the read speed of its SSD, of $\sim$700 MB/s).

Part II
Python

We will use the Python script as a reference. We will study its implementation details later, when we compare the Python implementation with the AWK and Perl scripts...

Below is our original implementation, in Python:

Figure 1: bucket.py

```
#!/usr/bin/python3

# This source code is under the BSD License
# Copyright 2022 Pierre S. Caboche

import sys
import re
import os
import codecs
sys.stdin = codecs.getreader('utf8')(sys.stdin.detach(), errors='
    ↪ ignore')
#sys.stdin.reconfigure(errors='ignore')

#mode='a+'   # append to existing file
mode='w+'    # write to file

outdir="out/python"
file_num = 0
filename = outdir + "/0000_BEGIN.sql"

pattern=r'^-- Table structure for table `?(?P<TABLE>\w+)`?'
parser = re.compile(pattern, re.IGNORECASE)

### Create the output directory if it does not exist
os.makedirs(outdir, exist_ok = True)

### File Handler (FH)
FH = open(filename, mode)

try:
  for line in sys.stdin :
    res = parser.match(line)
    if res:
      ### New table detected. Need to switch files
      FH.close()
      table = res.group('TABLE')
      file_num += 1
      filename = "{dir}/{num:04d}_{tbl}.sql".format(
        dir = outdir, num = file_num, tbl = table )

      ### Print the filename to stderr
      sys.stderr.write(filename + "\n")

      FH = open(filename, mode)
    FH.write(line)
finally:
  FH.close()
```

Make sure this script is executable by performing a:

```
chmod u+x bucket.py
```

A few things to notice…

Below are a few things I would like to point out regarding scripts in general, and this Python script in particular…

Specifying the interpreter

Unlike Windows, Linux does not rely on a file extension to determine how to run a program. As long as a file is marked as "executable", Linux will consider is as executable.

To determine which interpreter to use, the program loader looks at the first line of a script.

In a Linux script, the first characters of the first line are `#!`, also called *"shebang"* (because it comprises a ha*sh*tag and an exclamation mark, also known as *"bang!"* Hence the name *"shebang"*)

Once a *"shebang"* is detected, the program loader reads the rest of the first line to determine which interpreter to use (and with which parameters).

Note that we need to specify the full path of the interpreter executable. Usually, this will start with `/usr/bin`[1].

In this article, we will use the following at the beginning of our scripts…

For `python`:

```
#!/usr/bin/python3
```

For `gawk`:

```
#!/usr/bin/gawk -f
```

For `perl`:

```
#!/usr/bin/perl
```

For `julia`:

```
#!/usr/bin/julia
```

[1] `/usr/bin` contains binaries available to all users in multi-user mode, as opposed to `/bin` which is contains essential binaries required at boot up and single-user mode

Incrementing an integer

> Python doesn’t have the `++` operator for incrementation.
>
> In other languages, the misuse of the `++` operator might lead to unforeseen consequences, which *may* be the reason why `++` has been left out of Python.

If you try using the `++` operator in Python (like you might be doing in other programming languages), then you’ll be met with a syntax error, which can be confusing at first.

In any case, in Python, to increment an integer, you would need to do a `+= 1`:

```
# Python
file_num += 1
```

In the other languages (AWK and Perl), we will use the `++` unary operator to increment integers. For example in AWK:

```
# AWK
file_num++
```

Compiling the regular expression

Near the beginning of our script, we have the following instructions:

```
pattern=r'^-- Table structure for table `?(?P<TABLE>\w+)`?'
parser = re.compile(pattern, re.IGNORECASE)
```

Here, we “compile” the regular expression once, store the compiled expression in a variable (`parser`), and then use it again and again (`parser.match(line)`). This is to avoid compiling the regular expression every time we need it.

Reading from `stdin`

We read the content of `stdin` line by line, with the following loop:

```
  for line in sys.stdin :
  # loop instructions...
```

In this code, `sys.stdin` is an *iterator* over `stdin`, and our loop follows this pattern:

```
for <variable> in <iterator> :
# loop instructions...
```

Result

The result of execution is as follows (*note: some of the output was edited to fit in the page*):

```
[user@gen-8 test]$ time pv mysql_dump.sql | ./bucket.py
7.42GiB 0:14:55 [8.49MiB/s]
   ↪ [====================================================>] 100%

real    14m55.335s
user    14m29.250s
sys     0m28.531s
```

As you can see, our input file was 7.42 GiB in size, and Python took nearly 15 minutes to process it at an abysmal 8 MiB/s... (= 7420 / 15 / 60)

When running the script for the first time, `pv` showed a very low 10 MiB/s, and my first reaction was: *“did I do something wrong?”*. As we’ll discover later, this was not the case…

In the meantime (and while the Python script was still running…), I wanted to try another implementation. So I rewrote the script in AWK…

Part III
GNU awk

AWK is a domain-specific tool (with its own scripting language) for text manipulation. While the original AWK was created at Bell Labs in the 1970s, today there are several implementations (`nawk`, `gawk`, `mawk`, etc.)

`gawk` (the GNU implementation of AWK) offers many improvements over the original AWK like the ability to capture groups in *regular expressions* (which we will need in our script).

Usually on linux, the `awk` command is actually a symbolic link to some implementation of AWK, generally either `gawk` or `mawk`.

For example, in Fedora 35 we can see that `/usr/bin/awk` is a symbolic link to `/usr/bin/gawk` :

```
$ which awk
/usr/bin/awk
$ ls -l /usr/bin/awk
lrwxrwxrwx. 1 root root 4 Jul 22  2021 /usr/bin/awk -> gawk
```

`mawk` was designed for speed of execution, but it does not implement certain features (like group capturing, which we'll need).

In other words…

> On Linux, when using the "`awk`" command, you are never entirely sure which implementation you're actually using (it depends on how the system has been configured).
>
> It is therefore recommended to explicitly specify which flavour of AWK you intend to use (`gawk`, `mawk`, or other), because they don't all implement the same set of features.
> Different implementations of AWK are not always compatible with each other.

For this article, we'll be using `gawk`.

As seen in the previous part, our Python script was extremely slow. So I was curious to see if AWK could: 1) perform the task; and 2) fare better than Python in the performance department…

Here is the thing though: before that day, I had never written any AWK script. My experience with AWK was limited to the occasional one-liner command adapted from some sample code found online. Writing a full-fledged AWK script would be something new to me…
But while Python was still wrestling with our data, I would start looking into GNU awk and see what it could do.

The script

After a bit of research, this is the script I came up with. Incidentally, this is also the very first AWK script I wrote in my life…

Figure 2: bucket.awk

```
#!/usr/bin/gawk -f

# This source code is under the BSD License
# Copyright 2022 Pierre S. Caboche

BEGIN {
  IGNORECASE = 1;
  outdir = "out/awk"
  file_num = 0
  filename = outdir "/0000_BEGIN.sql"

  ### Create the output directory if it does not exist
  system( "mkdir -p " outdir )

  ### Empty first file
  printf("") > filename
}

{
  if ( match($0, /^-- Table structure for table `?(\w+)`?/, a) ) {
    ### This is a match. We need to change of bucket

    table_name = a[1]

    close(filename)
    file_num++
    filename = sprintf("%s/%04d_%s.sql", outdir, file_num,
       ↪ table_name)

    ### Print the filename to stderr
    print filename  > "/dev/stderr"

    print  > filename
  }
  else {
    print >> filename
  }
}

END {
  close(filename)
}
```

Make sure this script is executable, on Linux:

```
chmod u+x bucket.awk
```

Result

```
[user@gen-8 test]$ time pv mysql_dump.sql | ./bucket.awk
7.42GiB 0:00:08 [ 921MiB/s]
   ↪ [=====================================================>] 100%

real   0m8.253s
user   0m2.134s
sys    0m5.450s
```

Okay... so not only is the `gawk` script a lot more concise than its Python counterpart, but it's also a lot faster too!

The execution time was around 8 seconds (if the output folder exists and is empty). 15 seconds if the output folder still contained files from a previous execution (a few hundred of them)[2].

This is a whopping **60 to 100 TIMES FASTER** than Python!

By switching from Python to `gawk` we quite literally took our execution time from 15 **minutes** down to 15 **seconds**...

Now that we saw how AWK performed, let's try to understand how the script works...

GNU awk implementation

This is a rapid tutorial about `gawk` (with *some* comparisons to Python and Perl), which will takes us through the key parts of our script.

BEGIN and END

The BEGIN rule is run once, before any input record is read. We use it to initialise the script.
The END rule is run once, when we reach the end of the file/stream (and after all the input has been read and processed). We use it for cleanup purposes (i.e. closing the file handler for the last output file).

Note: even if we omitted the END rule entirely, `gawk` would close any open file handler at the end of the script execution. It is, however, a good habit to close any resource that we don't want to use anymore.

String concatenation

In AWK, string concatenation is done simply by putting variables (or literals) next to each other.

For example the line:

```
file = outdir a[1] ".sql"
```

...will concatenate the values of variable `outdir`, `a[1]` (element of indice 1 in array `a`), and the string literal `".sql"`; then assign the result to variable `file`.

[2] in other words, most of the execution time would be spent on things like: creating folders, emptying existing files... rather than actually processing the data.

This way of concatenating strings might seem odd at first (especially if you have some prior experience with other languages).

However, it is consistent with the behaviour of the `echo` command for the original AT&T UNIX®, for which AWK was initially developed.

String formating

For that, we use the ‘sprintf‘ function:

```
filename = sprintf("%s/%04d_%s.sql", outdir, file_num, a[1])
```

Creating the directory

We create a directory (and all sub-directories) by calling the system command to “make a path”: `mkdir -p`

The original `awk` from Bell Labs was designed to run on AT&T UNIX® machines. In this environment, performing system calls was not out of the ordinary.

However, this approach introduces some security issues. A directory is created through a system call, namely: `system( "mkdir -p " outdir )` where the value of `outdir` is read from the data file. With that in mind, it is trivially easy to craft a data file which will inject ”`some_dir; some_linux_command`” into the `outdir` variable, resulting in the command in question being executed on the system.

Something to keep in mind...

In all other languages in this article, directories are created through a dedicated API (not by blindly executing some Linux commands).

Matching regular expressions

We need to check if a regular expression is matched, use some capture groups, and branch if the expression is not matched. We use the `match` function for that:

```
if ( match($0, /^-- Table structure for table `?(\w+)`?/, a) ) {
  ### When expression is matched
  ...

} else {  ## (optional)
  ### When expression is not matched
  ...
}
```

The previous construct:

- allows to capture groups (stored in array `a`)
- allows the use of an `else` statement

If the `match` function succeeds, the result will be stored in array `a`:
`a[0]` contains the whole matched expression.
`a[1]` contains the 1st captured value.
`a[2]` contains the 2nd captured value.
And so on...

In Perl, the captured groups would be stored in variables `$1`, `$2`, `$3`, etc. but these have a different meaning in AWK (see below).

To my knowledge, `gawk` does not support named group capturing (unlike Python and Perl 5), so we will need to rely on group numbering only, which is harder to maintain.

If we didn't need to capture groups (just need to check that the line matches the regular expression), we could use the following construct (*note: needs to be inside a rule*):

```
if ( $0 ~ /some regular expression here/ ) {
  # What to do if the pattern matches
} else {  ## (optional)
  ### When expression is not matched
  ...
}
```

Finally, if we don't need to capture groups and don't need the `else` clause either, then we could use a rule:

```
/some regular expression here/ {
  # What to do if the pattern matches
}
```

So these are three ways to see if a line (or a string) matches a regular expression. The `match` function is the one we'll use, as we need to capture groups.

Note: later we will see how it is possible, and in many cases preferable, to get rid of `else` clauses entirely.

Variables `$0`, `$1`, `$2`, `$3`...

In AWK, the built-in variable `$0` contains the content of the current input line.

By default, AWK uses *whitespaces* and *tabs* as a field separator. The built-in variables `$1`, `$2`, `$3`, (and so on...) will contain the values for fields 1, 2, 3, (and so on...). This is very useful when dealing with *tabular files* (in which the data is presented in the form of rows and columns).

This meaning of such variables, however, is very different in Perl (as we'll see later...)

Printing to `stdout`

The `print` function is used to print variables or literals:

```
print "Hello!"
```

It is therefore possible to output the content of this line by printing `$0`:

```
print $0
```

...but this can be simplified to just:

```
print
```

Then it is also possible to redirect the output of `print` to `stderr` or to a file (as we'll see below...)

Printing to stderr

Here is an example where we print the value of ‘filename‘ to ‘stderr‘ :

```
print filename  > "/dev/stderr"
```

Note that "/dev/stderr" is in quotes (and therefore a string).

This tells gawk to output to the *standard error*, even on systems that don’t have the /dev/stderr file.

Writing to files

When switching files, we do the following:

```
print > file
```

Internally, this will open a file descriptor (for writing) to the file named file. This file descriptor will remain open until we explicitly close it, or will be closed automatically at the end of the script.

Because this is a single redirection (>), if the file exists its content will be overwritten (i.e. the file content will be erased). If the file does not exist, then the file will be created. Once the file is (created and) open, this instruction will output the current line to the file named file.

This is equivalent to:

```
print $0 > file
```

Later on, we’ll need to *append* more lines to the file (note that the file descriptor is still open at that point).
This is done using the double redirection >> :

```
print >> file
```

Finally, we will close the file descriptor with the following instruction:

```
close(file)
```

Then we’ll open the next file and the cycle starts again.

Opening many files

It is possible to keep the file descriptor open. At the end of the execution, AWK will close all open file descriptors automatically for us.

The problem with keeping too many files open is that we might exceed the maximum number of files that can be open simultaneously by one process. In linux, you can retrieve such (soft) limit with the command ulimit -Sn :

```
$ ulimit -Sn
1024
```

Here, we can see that the policy for the current user is to have no more than 1024 files open per process (this limit can be raised, but it requires super-user rights).
For this reason (and to avoid locking the file), it is recommended to limit the number of files open at the same time. Ideally, a file should be closed when no longer in active use.

Removing the `else` clause

The problem with the `if ... then ... [else if ...] else ...` construct is that it is prone to generating a long list of conditions which depend on each other. This makes the code harder to read and maintain.

To avoid this problem (in general, not just in AWK), the idea is to have a list of independent `if ...` constructs (without `else`), with some mechanisms to ignore the remaining `if` statements.

This can be done by using a function (and the `return` statement, to exit the function early):

```
function foo( arguments ) {

  if ( condition1 ) {
    ...
    return;  # early exit
  }

  if ( condition2 ) {
    ...
    return;  # early exit
  }
  ...
}
```

An AWK script can have the following structure

- a `BEGIN` block, which will execute at the beginning of the program, before any record is processed (initialisation)
- a series of separate rules, which will be evaluated in sequence for each record. The `next` statement can be used to stop processing the current record, and move to the next one.

  The approach can be used in some cases to avoid writing an AWK script as just one very long `if … else if … else …` block.
- an `END` block, which will execute after the last record has been processed

Regarding the `next` statement, here is an excerpt from the GNU AWK documentation:

> The `next` statement forces `awk` to immediately stop processing the current record and go on to the next record. This means that no further rules are executed for the current record, and the rest of the current rule's action isn't executed. (FSF, 2020)

Below is a new version of our script, rewritten to remove the `else` clause:

Figure 3: bucket2.awk

```
#!/bin/gawk -f

# This source code is under the BSD License
# Copyright 2022 Pierre S. Caboche

BEGIN {
  IGNORECASE = 1;
  outdir = "out/awk"
  file_num = 0
  filename = outdir "/0000_BEGIN.sql"

  ### Create the output directory if it does not exist
  system( "mkdir -p " outdir )

  ### Empty first file
  printf("") > filename
}

# Rule 1
{
  if ( match($0, /^-- Table structure for table `?(\w+)`?/, a) ) {
    ### This is a match. We need to change of bucket

    close(filename)
    file_num++
    filename = sprintf("%s/%04d_%s.sql", outdir, file_num, a[1])

    ### Print the filename to stderr
    print filename  > "/dev/stderr"

    print   > filename
    next      # Move to the next line. Ignore the remaining rules.
  }
}

# Rule 2
{
  print >> filename
}

END {
  close(filename)
}
```

Both versions have roughly the same performance.
Feel free to choose whichever version you find more readable.

Part IV
Perl

After rewriting the script from Python to `gawk`, I though that for the sake of comparison I might as well do it in Perl too, and see how it goes.

The script

And just like with the `gawk` script, what you're seeing here is my very first Perl program ever...

Figure 4: bucket.pl

```perl
#!/usr/bin/perl

# This source code is under the BSD License
# Copyright 2022 Pierre S. Caboche

use File::Path qw(make_path) ;

BEGIN: {
  $outdir = "out/perl" ;
  $file_num = 0 ;
  $filename = $outdir . "/0000_BEGIN.sql" ;

  # $mode = '>>' ;  # append
  $mode = '>' ;     # overwrite

  ### Create the output directory if it does not exist
  make_path($outdir, { chmod=>0770 }) ;

  ### File Handler ("FH")
  open(FH, $mode, $filename) or die $! ;
}

while (<>) {

  if ( $_ =~ /^-- Table structure for table `?(?<TABLE>\w+)`?/i ){

    ### This is a match. We need to change of bucket
    $table_name = $+{TABLE} ;

    close(FH) ;
    $file_num++ ;
    $filename = sprintf("%s/%04d_%s.sql", $outdir, $file_num,
       ↪ $table_name) ;

    ### Print the filename to stderr
    print STDERR $filename . "\n" ;

    open(FH, '>', $filename) or die $! ;
  }
  print FH ;
}

END: {
  close(FH) ;
}
```

Make sure this script is executable by performing a:

```
chmod u+x bucket.pl
```

Result

```
[user@gen-8 test]$ time pv mysql_dump.sql | ./bucket.pl
7.42GiB 0:00:08 [ 946MiB/s]
   ↪ [================================================>] 100%

real   0m8.035s
user   0m3.238s
sys    0m4.056s
```

The execution time is quite consistent with that of `gawk`, varying between 8 seconds (when the output folder exists but is empty) and 15 seconds (when the output folder contains hundreds of result files from a previous execution, which need to be emptied), with some outliers at around 18 seconds.

Now let's take a look at the Perl syntax...

Perl implementation

This is a rapid tutorial about Perl (with even more comparisons to Python and AWK), in which we'll go through the key parts of our script.

Semicolons

In Python and AWK, if a line contained only one statement, then it was possible to omit the semicolon (;) at the end of the line.

In Perl however, semicolon are mandatory at the end of each statement.

BEGIN: and END:

These are labels, which can be used as a target of a `goto` statement [3].

In our script, they are used to show the equivalent of the BEGIN and END rules from our `gawk` implementation, as well as to highlight the initialisation and clean up parts of the script.
Other than that (i.e. making the Perl script slightly clearer), those 2 labels could be removed and the script would behave exactly the same.

[3] `goto` statements are to be avoided. This is bad programming

Variables

Perl has different types of variables, and each type is represented by a character, called *sigil*, which appears in front of the variable name:

- scalar variables (e.g. integer, floating points, strings...): have their name preceded by the *sigil* `$`
- arrays: have their name preceded by the *sigil* `@`
- hash variables (key/value pairs): have their name preceded by the *sigil* `%`

Our script only mainly scalar variables (e.g. `$filename`, `$outdir`...), as well as a special variable: `$+` .

String concatenation

This is done with the `.` operator:

```
$filename = $outdir . "/0000_BEGIN.sql"
```

String formating

For that, we use the `sprintf` function:

```
$filename = sprintf("%s/%04d_%s.sql", $outdir, $file_num, $1) ;
```

This is very similar to the AWK syntax:

```
filename = sprintf("%s/%04d_%s.sql",  outdir,  file_num, a[1])
```

. . .but very different from the (recommended) way to format strings in Python 3:

```
filename = "{dir}/{num:04d}_{tbl}.sql".format(
dir = outdir, num = file_num, tbl = table )
```

Creating the directory

We create a directory (and all sub-directories) by using the `make_path` function:

```
use File::Path qw(make_path) ;
...
make_path($outdir, { chmod=>0770 }) ;
```

Reading from stdin

The diamond operator <> allows to read from the standard input (stdin). However, if a file name is passed as a parameter to the Perl script, then Perl will read from this file instead of stdin.

To read from stdin only , we may use <STDIN> instead of <>.

The input stream will be read line by line in a while loop. The content of each line (including the n line terminator) will be stored in the built-in variable $_ :

```
while (<>) {
  # The content of the line is stored in $_

  # Process the line here
  ...
}
```

Matching against regular expressions

Now that the content of the current line is stored in $_, it's time to try and match it with a *regular expression*.

This is done with the =~ operator.

```
if ($_ =~ /^-- Table structure for table `?(?<TABLE>\w+)`?/i) {
  # Do something if the match is successful
  ...
}
```

In this regular expression, we use a named capturing group (available since Perl 5, and borrowed from Python).
A named capturing group in Perl is of the form (?<group_name>expression). The captured values go into the special variable $+, which contains key/value pairs.

In our regular expression, we have the following (?<TABLE>\w+) . Here, the group is named 'TABLE'. If the match succeeds, the captured value can be retrieved using: $+{TABLE}.

Captured values are also stored in variables $1, $2, $3...where $1 is the 1st captured value, $1 the 2nd, and so on...

> The problem when working with *numbered* capturing group (as opposed to *named* capturing group) is that if your regular expression changes (especially if you add or remove parentheses) then the numbering might change too. This can make it very difficult to work on complex regular expressions.
>
> For this reason, it is advisable to use named capturing group when supported by a given language or tool.

Named capturing group in Python are of the following form:
(?P<group_name>expression) .

To my knowledge, gawk does not seem to support named capturing group, so we used numbered capturing group instead.

Variables $1, $2, $3... and $0

As we've just seen, the variables $1, $2, $3... can be used to retrieve the values of capturing group from regular expressions.

The variable $0, however, has a totally different purpose as it contains the name of the script being executed (just like in shell scripts).

Printing to stdout

The print function is used to print variables or literals:

```
print "Hello " . $name . "\n" ;
```

Note that in Perl, the print instruction does not automatically add a new line at the end (unlike in AWK or Python, where it does). So in Perl you need to specify when you want to print "n" .

In Perl, it is therefore possible to output the content of the current line (as read by the diamond <> operator) by printing $_ :

```
print $_ ;
```

...but like with AWK, this can be simplified to just:

```
print ;
```

Printing to stderr

To print to the standard error, just call print with the STDERR handle:

```
print STDERR $filename . "\n" ;
```

STDERR is the handle for the standard error.
There exists a handle for the standard output (STDOUT), but calling
print STDOUT is exactly the same as just calling print .

Finally, we can pass a file handle (which we'll call FH) to print to a file, as we'll see in the next paragraph...

Writing to files

The first thing to do is to open the file with the function:

```
open(file_handler, mode, path)
```

where:
path is the path to the file to open.
mode is one of the following:

Table 3: File modes in Perl

mode(*)	Description
"<" or "r"	Opens for reading. The file pointer is placed at the beginning of the file.
"+<" or "r+"	Opens for both reading and writing. The file pointer is placed at the beginning of the file.
">" or "w"	Opens for writing only. Overwrites the file if the file exists. If the file does not exist, creates a new file for writing.
"+>" or "w+"	Opens for both writing and reading. Overwrites the file if the file exists. If the file does not exist, creates a new file for writing.
">>" or "a"	Opens for appending. If the file exists, the file pointer is at the end of the file. If the file does not exist, creates a new file for writing.
"+>>" or "a+"	Opens for both appending and reading. If the file exists, the file pointer is at the end of the file. If the file does not exist, creates a new file for writing.

(*)add ":raw" for binary mode.

Sources: Rohit (2020), Farrel (2016), GFG (2019)

Once opened, the file will be represented by a file_handler, which can be used in the same way as handler such as STDERR or STDOUT.

For example, we open a file (for overwriting) like this:

```
open(FH, ">", $filename) ;
```

Then we can use our file handler FH to write to the file:

```
print FH "some string" ;
print FH $some_variable ;

### Print the line read from <>
print FH $_ ;
print FH ;
```

Finally, we'll need to close the file with:

```
close(FH) ;
```

File modes in Python

Here is a quick comparison of the file modes in Python and Perl:

Table 4: File modes in Python vs. Perl

File `mode` in Python	File `mode` in Perl	Description
`"r"`	`"<"` or `"r"`	See Table 3
`"r+"`	`"+<"` or `"r+"`	
`"w"`	`">"` or `"w"`	
`"w+"`	`"+>"` or `"w+"`	
`"a"`	`">>"` or `"a"`	
`"a+"`	`"+>>"` or `"a+"`	
`"rb"`	`"<:raw"` or `"r:raw"`	Same as `"r"`, but in binary format.
`"rb+"`	`"+<:raw"` or `"r+:raw"`	Same as `"r+"`, but in binary format.
`"wb"`	`">:raw"` or `"w:raw"`	Same as `"w"`, but in binary format.
`"wb+"`	`"+>:raw"` or `"w+:raw"`	Same as `"w+"`, but in binary format.
`"ab"`	`">>:raw"` or `"a:raw"`	Same as `"a"`, but in binary format.
`"ab+"`	`"+>>:raw"` or `"a+:raw"`	Same as `"a+"`, but in binary format.
`"x"`	n/a	Open for exclusive creation; fails if the file already exists (Python 3)

Rohit (2020), Farrel (2016)

Part V
Julia

The last language we will evaluate is Julia. It is the most recent on our list, first appearing in 2012 (for reference, AWK first appeared in 1977, Perl in 1988, and Python in 1991).

Julia took some inspiration from a variety of languages released before it (from Matlab to Python, Ruby, Perl, and more...) with a focus on speed, as explained by Bezanson et al. in their article *"Why We Created Julia"*:

> *"We want a language that's open source, with a liberal license. We want the speed of C with the dynamism of Ruby. We want a language that's homoiconic, with true macros like Lisp, but with obvious, familiar mathematical notation like Matlab. We want something as usable for general programming as Python, as easy for statistics as R, as natural for string processing as Perl, as powerful for linear algebra as Matlab, as good at gluing programs together as the shell. Something that is dirt simple to learn, yet keeps the most serious hackers happy. We want it interactive and we want it compiled."* — Bezanson et al. (2012)

Being a relatively recent language, Julia currently has considerably less adoption than other languages like Python.

Julia provides a lot of interesting features, like: distributed and parallel computation, interoperability with other languages (like Python, C, MATLAB...), a great support for Unicode, a built-in package manager, support for metaprogramming and Lisp-like macros, and many more... (csewo, 2020) (w3adda, 2022) (Boudreau, 2020)

It is to be noted that Julia currently has a number of bugs, and is in constant evolution. (Apgar, 2022)

Now we see how our script looks when rewritten in Julia, and later we will compare its performance with other languages on our list...

Here is our Julia script:

Figure 5: bucket.jl

```
#!/usr/bin/julia

# This source code is under the BSD License
# Copyright 2022. Pierre S. Caboche

using Base.Filesystem
using Printf

function put_in_buckets(outdir)

  file_num = 0
  filename = outdir * "/0000_BEGIN.sql"

  regex = r"^-- Table structure for table `?(?<TABLE>\w+)`?"i
  mkpath(outdir, mode=0o777)

  mode="w+"
  FH = open(filename, mode)

  for line = eachline()
    m = match(regex, line)

    if !isnothing(m)
      table_name = m[:TABLE]

      close(FH)

      file_num += 1
      filename = @sprintf("%s/%04d_%s.sql", outdir, file_num,
         ↪ table_name)

      ### Output the filename to logger
      @info filename

      FH = open(filename, mode)
    end

    write(FH, line, "\n")
  end

  close(FH)
end

put_in_buckets("out/julia")
```

The syntax of Julia is quite easy to follow, especially if you are already used to a language like Python, with a few differences...

Variables scope

In the Julia script, we've put the implementation in a function (`put_in_buckets`) which we later call. This has to do with how Julia handles variable scopes.

It is possible to use global variables in Julia, but it is not advisable. Indeed, if we didn't put out implementation inside of a function, then within the different code blocks (i.e. `for`, `if`), Julia would create local variables with the same name as the global variables.

This means we would need to use the `global` keyword whenever we need to assign a value to a global variable (in our script, this would mean using the `global` keyword for variables: `file_num`, `filename`, and `FH`).

Without the `global` keyword, Julia will refer to a local variable instead (which is not correct), and issue some warnings and errors, as illustrated below:

```
 Warning: Assignment to `file_num` in soft scope is ambiguous
    ↪ because a global variable by the same name exists: `
    ↪ file_num` will be treated as a new local. Disambiguate by
    ↪ using `local file_num` to suppress this warning or `global
    ↪ file_num` to assign to the existing global variable.
 @ path/to/bucket.jl:29
ERROR: LoadError: UndefVarError: file_num not defined
Stacktrace:
[1] top-level scope
@ path/to/bucket.jl:29
in expression starting at path/to/bucket.jl:21
```

In other words, in Julia it is better to avoid modifying global values, and organise the code in functions.

Code blocks

Regarding code blocks, languages like AWK or Perl use curly brackets (`{` and `}`) to define code blocks, and Python relies on indentation for that purpose.
Julia has a different approach...

In our script, keywords like `function`, `for`, or `if` marks the beginning of a code block. What follows will be evaluated by Julia, and the end of a code block will be indicated by the keyword `end`.

So for example, in a `if` block where the condition spans over several lines, we would need to put the condition in parenthesis. The instructions then follow, up until the `end` keyword is met:

```
if (    <condition1>
    && <condition2>
    && <condition3>
)
  <instructions>
end
```

Semicolons

Unlike with Perl, statements do not always need to be terminated by a semicolon in Julia. Where it makes sense, Julia will consider the end of line to mark the end of a statement (otherwise it will continue the evaluation on the next line).

Reading line by line

All four implementations read the file line by line, whether it be using a `for` loop (Python, Julia), a `while` loop (Perl), or a totally different mechanism (AWK).

In Julia:

```
for line = eachline()
```

Is this a bug or a feature?

One thing worth noting with the Julia implementation is that, unlike with other languages, the string value does not include the end of line character (“`\n`”). Therefore, when writing to the file handler (`FH`) we need to specifically add an end of line:

```
write(FH, line, "\n")
```

Incrementing an integer

Just like Python (and unlike AWK and Perl), Julia doesn’t have the `++` operator for incrementation.

That’s why we perform a `+= 1` to increment an integer:

```
# Julia
file_num += 1
```

Regular expressions

Julia uses Perl-compatible regular expressions, as provided by the PCRE library.

This means that Julia’s regular expressions support a lot of features, including named groups (not supported by AWK), which makes working with regular expressions easier.

In our regular expression, we define the “TABLE” named group, which is later extracted (from `m`, our list of matches) like this:

```
table_name = m[:TABLE]
```

Logger

In our Julia implementation, we do not display the progress to the standard output, but instead we register the computation progress using the Logging module, which allows to keep track of events with different gravity levels: debug, info, warn, error.

We are using the “info” level to display the name of the file our script is writing to:

```
### Output the filename to logger
@info filename
```

By default, events of level “info” will be displayed in the standard output. This can be configured programmatically (as described in the “Logging” section of the Julia documentation).

Part VI
Other technical considerations

Overwriting the files

In all implementations (Python, AWK, Perl. Julia), we overwrite the file whenever we open a new one.

This was done out of convenience for our particular example, to verify our results more easily (see Part VIII). In our case:

- we needed to make sure that our output files are overwritten each time a script is executed again (i.e. we do not want the result of previous executions in our output files), even if we forget to purge our output files before each execution
- we knew that our input data (a MariaDB database dump file) would not switch back and forth between buckets (i.e. bucket A, then bucket B, then back to bucket A) so our scripts would never write to the same output file twice.

For these reasons, we decided to *overwrite* our output files, instead of *appending* to them. We also made sure to *number each of our files*, so that we easily reconstruct the input from the output files (as described in Part VIII: *"Verifying our results"*).

That being said, depending on your use-case you might want to modify our scripts in order to write to the same output file more than once. To do so, you will need to take the following precautions:

1. first of all, before executing the script you would need to clear the content of existing output files, but *only if necessary* (there are cases when you would want to keep the result of previous executions and append data to existing files)

   In any case, it is *your responsibility* to empty (or remove) the output files in-between executions of the scripts.

2. then, you would need to modify the scripts to *append* data to files instead of *overwriting* files

   - in the Python script, this is done by changing the file open `mode` from `"w+"` to `"a+"`
   - in the `gawk` script, this is done by changing the `print > file` to `print >> file`
   - in the Perl script, this is done by changing the file open `mode` from `">"` to `">>"`

Of course, we could add an option for the scripts to *append to* the output files instead of overwriting them (similar to the *-a* option of the `tee` command), but this goes beyond the scope of this article.

Part VII
Python issues

So… why is Python so slow?

So far we haven’t seen why the Python script was so much slower than the `gawk` implementation. To try and answer this question, we’ll use a Python profiler (*cProfile*).
The output of *cProfile* is a bit large, so I’ve removed some of it (*the removed parts do not affect the conclusion*). Some of the output results like high timings have been emphasised (*emphasis mine*):

Figure 6: Profiling our Python script

```
[user@gen-8 test]$ time pv mysql_dump.sql | python3 -m cProfile ./bucket.py
7.42GiB 0:14:57 [8.47MiB/s] [===========================================================>] 100%
17332856 function calls (17332841 primitive calls) in 897.093 seconds

Ordered by: standard name

ncalls  tottime  percall  cumtime  percall filename:lineno(function)
1    0.000    0.000    0.000    0.000 __init__.py:43(normalize_encoding)
1    0.000    0.000    0.000    0.000 __init__.py:70(search_function)
828    0.001    0.000    0.002    0.000 _bootlocale.py:33(getpreferredencoding)
1    0.000    0.000    0.000    0.000 codecs.py:1005(getreader)
828    0.000    0.000    0.000    0.000 codecs.py:186(__init__)
828    0.000    0.000    0.000    0.000 codecs.py:260(__init__)
546234    0.048    0.000    0.048    0.000 codecs.py:276(reset)
828    0.001    0.000    0.001    0.000 codecs.py:309(__init__)
546234    0.186    0.000    0.234    0.000 codecs.py:327(reset)
1    0.000    0.000    0.000    0.000 codecs.py:423(__init__)
1269646    5.147    0.000   15.599    0.000 codecs.py:451(read)
546235   40.593    0.000  891.533    0.002 codecs.py:531(readline)
546235    0.187    0.000  891.720    0.002 codecs.py:642(__next__)
1    0.000    0.000    0.000    0.000 codecs.py:650(__iter__)
1    0.000    0.000    0.000    0.000 codecs.py:94(__new__)
2    0.000    0.000    0.000    0.000 enum.py:313(__call__)
2    0.000    0.000    0.000    0.000 enum.py:631(__new__)
(...)
2    0.000    0.000    0.000    0.000 {method 'rfind' of 'str' objects}
1    0.000    0.000    0.000    0.000 {method 'rstrip' of 'str' objects}
2343068  834.227    0.000  834.227    0.000 {method 'splitlines' of 'str' objects}
546234    4.491    0.000    4.725    0.000 {method 'write' of '_io.TextIOWrapper' objects}


real  14m57.128s
user  14m52.051s
sys   0m32.191s
```

From the output of *cProfile*, we can see that the `readline` function takes the longest time.

The `readline` function is called when we do a:

```
for line in sys.stdin :
```

This type of loop goes through the element of an iterator, and is of the form:

```
for <variable> in <iterator> :
```

The profiler doesn’t show it, but the memory consumption of the script is very low (at around 10 MB). As expected, the input is read line by line. What was not expected, however, was the fact that this operation would be so slow in Python.

When looking for solution online, I saw many people suggesting that `readlines` (plural) is a lot faster than `readline`. However, this forces to put the whole file in memory first, which is not viable (because we might deal with streams, or very large files).

This approach might work if you can afford to put all your data in memory before working on it, but it starts to fall apart when you can't.
Some Data Analysts may be fine with this approach; they would usually use smaller data sets to train / validate/ test Machine Learning models, and then use those models to make predictions on small data sets. Data Engineers, on the other hand, often need to clean / filter large amounts of data for use by Data Analysts (e.g. separate the data in smaller files like in our example). They might be better off using AWK to process such large files.

That being said, speed is not the only problem we've encountered with Python. Indeed, there is worse...

Problems with UTF-8 encoding

For the sake of clarity I haven't mentioned another problem that I met with Python. Now is the time to talk about it.
The files we are trying to process are MySQL database backup (dump) files. Some of the data they contain might be of type binary. This has some consequences... When trying to process such file with Python, the following error occurs:

```
UnicodeDecodeError: 'utf-8' codec can't decode byte 0xe2 in
    ↪ position 3987: invalid continuation byte
```

To get around this problem, some comments online suggested to do the following:

```
import codecs
sys.stdin = codecs.getreader('utf8')(sys.stdin.detach(), errors='
    ↪ ignore')
```

Although it removed the previous error and allowed to execute the script (albeit in an excruciatingly long time), some of the data was missing from the output (as implied by `errors='ignore'`, we are simply ignoring the errors thrown by the `utf8` codec).

This isn't good.

Another solution would be to read the input as binary, but then it's not possible to call the `readline` function (directly or indirectly). We would then have to implement our own buffer, detect the ends of lines, apply our regular expression only when needed (at the beginning of a line)...This makes the script much more complicated, which defeats the purpose of using Python, a language often hailed for its simplicity.

In the end, we *might* eventually achieve *correctness*, but at the detriment of *simplicity* (and we would most likely not achieve *speed* either...).

I didn't explore the Python solution any further. I had initially chosen Python because it was supposed to be simple, but then new problems surfaced (utf-8 error, speed...), which required finding new solutions or workarounds.

That's when I thought it would be better to take a step back, try a solution in AWK (a language I knew almost nothing about), and see how it compares to Python...
Later I would also try other solutions in Perl and Julia, which both confirmed my previous results.

Part VIII
Verifying our results

We need to verify that our output files are correct, which means that if we were to concatenate them back together (and in the same order), the result would be identical to the original files.

Our approach

From the beginning, we took care to separate our output files by implementation, so it will be easier to compare the results.
We also numbered our files, so we know in which order they were created. Our file names start with `0000`, `0001`, `0002`, and so on... This way we can sort the files alphabetically, and it will follow the order in which they where created.

> We cannot compare the file contents directly with one another because **the files are too big**, so a tool like `diff` would quickly run out of memory.
>
> Instead, we are going to compute some checksum (with a command like `sha512sum`) and then compare those checksums with each other. Any minor change in the data will result in widely different checksums, and the risk of different data having the same checksum is infinitesimally low.
>
> Also, we are not going to concatenate the files back together (in yet another temporary file), but instead use a linux command (like `cat`or `pv`) to read the files one by one, and feed this stream of data to `sha512sum` using pipes.

So first, we need to calculate the checksum of the original file (*note: some of the output was edited to fit in the page*):

```
[user@gen-8 ~]$ time pv in/mysql_dump.sql | sha512sum
7.42GiB 0:00:12 [ 589MiB/s]
   ↪ [====================================================>] 100%
1e3429096ec3f412c2eecca81eafd82a9237e7cd03f070cace01b28e33a2445
072af2773aee6f18c415f3e98162d55b0df3662e5dbf3df18855584823d5176
b5  -

real  0m12.890s
user  0m11.841s
sys 0m2.833s
```

Then we can compare with the checksum of our output files for each implementation:

```
[user@gen-8 test]$ pv out/perl/* | sha512sum
7.42GiB 0:00:13 [ 584MiB/s]
   ↪ [====================================================>] 100%
1e3429096ec3f412c2eecca81eafd82a9237e7cd03f070cace01b28e33a2445
072af2773aee6f18c415f3e98162d55b0df3662e5dbf3df18855584823d5176
b5  -

[user@gen-8 test]$ pv out/awk/* | sha512sum
7.42GiB 0:00:13 [ 572MiB/s]
   ↪ [====================================================>] 100%
1e3429096ec3f412c2eecca81eafd82a9237e7cd03f070cace01b28e33a2445
072af2773aee6f18c415f3e98162d55b0df3662e5dbf3df18855584823d5176
b5  -

[user@gen-8 test]$ pv out/julia/* | sha512sum
7.42GiB 0:00:13 [ 572MiB/s]
   ↪ [====================================================>] 100%
1e3429096ec3f412c2eecca81eafd82a9237e7cd03f070cace01b28e33a2445
072af2773aee6f18c415f3e98162d55b0df3662e5dbf3df18855584823d5176
b5  -

[user@gen-8 test]$ pv python/* | sha512sum
7.40GiB 0:00:12 [ 592MiB/s]
   ↪ [====================================================>] 100%
48d32c2dd1badac05aca840a1e61d97383ea5514abc9b0b4c116144a2b7a492
4ca487bccd9218c2f3300961849ff7c8533f35eefbfa0bfcd62fca311f69b9d
a5  -
```

What we can see from those results is that the checksums for the output of the AWK Perl and Julia implementations are identical to that of the original file (i.e. the content is correct).

However, the checksums for the output of the Python script is completely different. Also, the total size of the output data for Python is smaller than the original. Because of the encoding problem we encountered earlier, **some data was lost**.

Margins of error

As we have seen previously, our AWK, Perl, and Julia scripts usually executed in only a few seconds.

However, there were times when those scripts would take longer to execute, probably because some other background processes which interfered with our script execution in one way or another (e.g. by accessing the file system at the same time). Although a few seconds are barely noticeable over the span of several minutes, it becomes a problem when the time-frame is smaller (i.e. every 1 second delay on a process which usually executes in 5 seconds represents a margin of error of 20%, which is enormous). To counteract this issue, I executed the scripts several times, to get a better overall average execution time.

That being said, no matter how you measure it, Python still remains 1 or 2 orders of magnitude slower than all the others scripting languages tested...

Part IX
Results and Conclusion

Here is a summary of what we've learned about each language, and how they performed (on the older machine, with an Intel® Core™ i5-8259U processor and PCI-Express 3 NVMe SSD):

- AWK was initially developed at Bell Labs in the 1970s for the original AT&T UNIX®. Its modern GNU implementation, `gawk`, implements features such as regular expressions *group capturing* (but not *named* group capturing).

  GNU awk 5.1.0 processed our file in about 9 seconds (with some outliers. The very worst cases took less than 15 seconds).

- Perl was initially designed in 1987 by Larry Wall, but now the name "Perl" refers to Perl 5, first released in 1994 after a near complete rewrite of the interpreter, and the addition of new features (including *named group capturing* in regular expressions, a feature inspired by Python).

  Perl 5.34.1 processed our file in about 9 seconds (with some outliers. The very worst cases took less than 15 seconds).

- Python is a slightly more modern language, first appearing in 1991.

  Python 3.10.5 processed our file in around 15 minutes, which is between 60 and 100 times longer than either `gawk` or Perl…

  Also, due to some problems with how Python handles UTF-8, some of our data was lost. Trying to fix the issue is not an easy task, and it was actually simpler to rewrite the script in either `gawk` or Perl (despite initial the learning curve).

- Julia is the most recent language, first appearing in 2012.

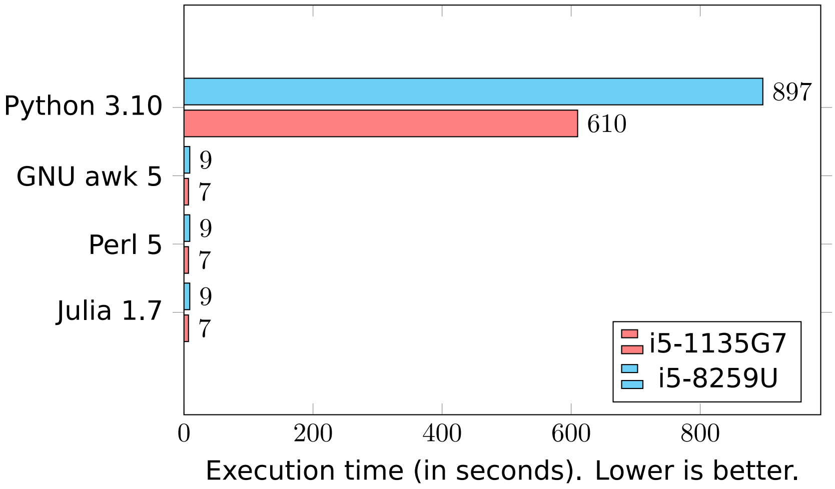
  Julia is a young but promising language, aimed to be easy to learn, very versatile, and fast. Julia's goal is to be usable for a wide array of applications, including general programming, file processing, statistics, and more…

  Julia 1.7.3 processed our file in about the same time as AWK and Perl (around 9 seconds).

Later, I repeated the tests on new computer with a faster CPU (Intel® Core™ i5-1135G7), a faster (PCI-Express 4) NVMe SSD, and a freshly installed Linux.

Below are the results:

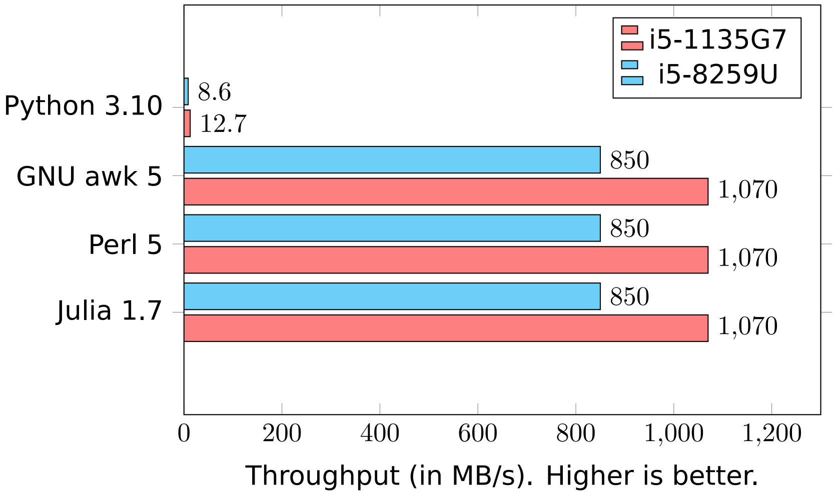
Figure 7: Speed comparison of Python, AWK, and Perl



On the 1135G7, the scripts in AWK, Perl, and Julia all took between 5 and 9 seconds to execute. This difference of speed can be attributed to external factors (i.e. systems processes which run in the background and interfere with the results).
Meanwhile on the same computer, the Python script took 10 minutes to execute. Given that the Python script takes so long to run, a slowdown of a few seconds is barely noticeable...

Another way to look at the results is by comparing the processing speed of each language (by dividing the input size by the total execution time).

Figure 8: Processing speed comparison of Python, AWK, and Perl



In figure 8, the difference of speed between Python and the other two languages is really striking!

For reference, the theoretical maximum throughput of SATA 3 is 600 MB/s. Both AWK and Perl process the data at a higher rate than that, and we are limited only by the CPU's single thread processing speed, not by the speed of NVMe SSD (whether it be PCI Express 3 or PCI Express 4).

Also worth noting:

- We used processors of different generations: an 8th generation Intel® Core™ (i5-8259U), and an 11th generation Intel® Core™ (i5-1135G7).

  Between the two processors, released a little over 2 years apart (one in Q2'2018, the other in Q3'2020), we observe a 33% difference in performance throughout the board. This is consistent with online benchmarks, which show a *single core* performance[4] of 37% between the two processors (UserBenchmark, 2021).

- Another notable difference between those two computers was the use of a much better SSD on the more recent machine.

  This did not impact the results (i.e. all our scripts were able to read/write as much data as they could process. Even the slower SSD did not constitute a bottleneck[5])

Key takeaways

Python has a reputation for being a very slow scripting language.

In this paper, we showed that this reputation is highly deserved.

In fact, the very reason why we looked at other scripting languages (AWK, Perl, and Julia) is because Python was so slow that it became impractical to use in our particular use-case. To add insult to injury, we also ended up with some corrupted data in our output files because of the way Python handles UTF-8 streams.

Now let's review some of Python's strong points, and see if they still hold up today:

- *"Python is cross-platform"*

  For a long time, this was a major advantage of Python: the ability to develop a script on one system, while knowing it should be able to run on other systems.

[4] our scripts don't make use of multiple cores

[5] in some previous test on another machine, the SSD read/write speeds were below 700 MB/s, which created a bottleneck that affected the performance of the `gawk` and Perl scripts. The Python script was not impacted because it could barely process 12 MB/s.

This has been less relevant today, especially on Windows, thanks to Windows Subsystem for Linux which allow to run any Linux scripts on a Windows machine, with some caveats[6]

- *"Python is very popular"*

  Being popular means that there exists a lot of resources online about Python, a great number of engineer will have Python on their résumé, and many job postings will also mention Python as a desired skill.

  What those numbers don't show is the number of companies which require Python because it is part of their technical stack, but are actively looking to move away from it for a reason or another.

  I do not have exact figures on the matter, only empirical data. That being said, it is good to ask how Python is used in a company, and if they are considering alternatives.

- *"Python provides a lot of libraries, especially for data analysis"*

  In that particular domain, one scripting language to watch is Julia.

  Not only is Julia a general-purpose language very well versed in numerical analysis, but as we saw in this article, Julia is really fast too.

[6] notably, it is difficult to access a Windows service (e.g. a database) from the Linux Subsystem. In essence, WSL2 creates a Linux virtual machine which runs on a Windows host. Such virtual machine has access to the host's file system (through `/mnt`), but is seen by Windows as a separate machine, which you need to grant access to (by configuring the firewall, and allow the service to accept connection from that machine). It is not straightforward.

Conclusion

In this article, we showed some of the shortcomings of Python, and explored different alternatives.

Not only was Python frustratingly slow for processing large data files (for the purpose of cleaning up data, sorting, modifying the format, etc.), but it also corrupted some of our data (the non-valid UTF-8 characters, i.e. the binary data which was part of our file).
We tried different alternatives, which all gave much better results for that particular use-case.

Here is a summary of the different languages we tried:

AWK

AWK is a very specialised language (designed mainly for text processing) but is very good at what it does! Its syntax is simple, even if requires a bit of getting used to it.

I personally use GNU Awk rather frequently, when dealing with large text file. I appreciate its simplicity and its speed.

Perl

Perl is more advanced than AWK, offering similar speed, and more features.

One of the problems of Perl is its syntax. Perl usually offers multiple ways to handle a problem, but they often relies on some specific syntax, making harder to learn (on top of the fact that different Perl programmers may have different coding styles)

Julia

Julia is probably one of the most criminally under-appreciated languages of the moment.

Julia has a syntax that is generally easy to follow, provides access to a great number of libraries, all while delivering high execution speed.

Of all the languages tested, Julia objectively possesses the most qualities:

- Julia's syntax is easy to learn (easier than Perl)
- Julia is very versatile (unlike AWK)
- Julia natively possesses many of the same features as Python (Julia can call Python libraries with PyCall.jl, or make direct calls to C and Fortran libraries)
- Julia offers great performance (on par with AWK and Perl)

AWK still remains a very good option for file processing, because of its very simple syntax. For more complex tasks however, Julia should be really high on your list of languages to consider.

If you intent to start a Python project but have the possibility to choose another language, you might want to give Julia a try. It is very likely that Julia would be able to perform the same tasks as Python, but better.

Hopefully this article will have given you a better understanding of the characteristics of each language, and that the examples provided would help you quickly get started and familiarise yourself with the syntax and features.

Going further

All the scripts used in this article are provided under the BSD license (please see the “Legal” section), which is very permissive.

This allows you to:

1. quickly get started with processing files in Python, AWK, Perl, or Julia
   (a) modify the scripts for your own data-processing needs
2. reproduce the experiments detailed in this article (i.e. separate data into different buckets) and see the results for yourself
3. extend the experiment to include other languages

Regarding point 1 (quickly getting started), our different examples are easy to follow yet cover useful programming concepts (variables, loops, conditions, I/O, files, regular expressions, etc.). Also, comparing the different implementations allows us to focus on what makes each language unique, and the different approaches they take in solving certain problems.

As far as point 2 is concerned (reproducing the experiments), please note that I am not at the liberty to provide you with any dataset. You will need to use your own database dump files.

A good dataset should contain a mix of big and small tables (in terms of number of records), wide and narrow tables (in terms of number of fields), wide and narrow data (in terms of string lengths), and potentially some binary data.

Finally concerning point 3, this article started as a comparison between Python and AWK for the processing of large text files. Perl and Julia were added later on, because they are well suited for the processing of files.

Comparing more languages would necessitate some changes in the methodology, notably:

- a standardised dataset would have to be created (see point 2)
- there is a need to distinguish between scripting languages (whose code can be easily modified) and compiled languages (which usually have the advantage of speed)
- one language may have more than one reference implementation. In that case, it would be necessary to compare the different alternatives

If that were to happen, this would require a separate paper, more focussed on comparing execution speed than analysing the differences in features and syntax between the different languages.

References

Apgar, V. (2022). Is julia actually right for you? https://towardsdatascience.com/is-julia-actually-right-for-you-b2c003d7cddf.

Bezanson et al. (2012). Why We Created Julia. *julialang.org*. https://www.perl.com/pub/2008/04/23/a-beginners-introduction-to-perl-510.html/.

Boudreau, E. (2020). Julia's most awesome features. https://towardsdatascience.com/julias-most-awesome-features-be51f798f140.

csewo (2020). Features of julia. https://www.cseworldonline.com/articles/features-of-julia.php.

Farrel, D. (2016). How to parse binary data with Perl. *Perl.com*. https://www.perl.com/article/how-to-parse-binary-data-with-perl/.

FSF (2020). Gawk: Effective AWK Programming. *Free Software Foundation*. https://www.gnu.org/software/gawk/manual/html_node/Next-Statement.html.

GFG (2019). Perl | File Handling Introduction. *GeeksForGeeks*. https://www.geeksforgeeks.org/perl-file-handling-introduction/.

Rohit (2020). Python file modes | Open, Write, append (r, r+, w, w+, x, etc). https://tutorial.eyehunts.com/python/python-file-modes-open-write-append-r-r-w-w-x-etc/.

UserBenchmark (2021). i5-8259u vs i5-1135g7. https://cpu.userbenchmark.com/Compare/Intel-Core-i5-8259U-vs-Intel-Core-i5-1135G7/m543736vsm1286124.

w3adda (2022). Julia features. https://www.w3adda.com/julia-tutorial/julia-features.

List of Figures

List of Tables