Persistent-Memory gawk User Manual

16 August 2022
gawk version 5.2
pm-gawk version 2022.08Aug.03.1659520468 (Avon 7)

Terence Kelly
tpkelly@eecs.umich.edu
tpkelly@cs.princeton.edu
tpkelly@acm.org
http://web.eecs.umich.edu/~tpkelly/pma/
https://dl.acm.org/profile/81100523747

1 Introduction

GNU AWK (gawk) 5.2, expected in September 2022, introduces a new *persistent memory* feature that makes AWK scripting easier and sometimes improves performance. The new feature, called "pm-gawk," can "remember" script-defined variables and functions across executions and can pass variables and functions between unrelated scripts without serializing/parsing text files—all with near-zero fuss. pm-gawk does *not* require non-volatile memory hardware nor any other exotic infrastructure; it runs on the ordinary conventional computers and operating systems that most of us have been using for decades.

The main gawk documentation[1] covers the basics of the new persistence feature. This supplementary manual provides additional detail, tutorial examples, and a peek under the hood of pm-gawk. If you're familiar with gawk and Unix-like environments, dive straight in:

- Chapter 2 [Quick Start], page 2, hits the ground running with a few keystrokes.
- Chapter 3 [Examples], page 3, shows how pm-gawk streamlines typical AWK scripting.
- Chapter 4 [Performance], page 6, covers asymptotic efficiency, OS tuning, and more.
- Chapter 5 [Data Integrity], page 14, explains how to protect data from mishaps.
- Chapter 6 [Acknowledgments], page 15, thanks those who made pm-gawk happen.
- Appendix A [Installation], page 15, shows where obtain pm-gawk.
- Appendix B [Debugging], page 15, explains how to handle suspected bugs.
- Appendix C [History], page 16, traces pm-gawk's persistence technology.

You can find the latest version of this manual, and also the "director's cut," at the web site for the persistent memory allocator used in pm-gawk:

http://web.eecs.umich.edu/~tpkelly/pma/

Two publications describe the persistent memory allocator and early experiences with a pm-gawk prototype based on a fork of the official gawk sources:

- https://queue.acm.org/detail.cfm?id=3534855
- http://nvmw.ucsd.edu/nvmw2022-program/nvmw2022-data/nvmw2022-paper35-final_version_your_extended_abstract.pdf

Feel free to send me questions, suggestions, and experiences:

tpkelly@eecs.umich.edu (preferred)
tpkelly@cs.princeton.edu
tpkelly@acm.org

[1] See https://www.gnu.org/software/gawk/manual/ and `man gawk` and `info gawk`.

2 Quick Start

Here's pm-gawk in action at the bash shell prompt ('$'):

```
$ truncate -s 4096000 heap.pma
$ export GAWK_PERSIST_FILE=heap.pma
$ gawk 'BEGIN{myvar = 47}'
$ gawk 'BEGIN{myvar += 7; print myvar}'
54
```

First, `truncate` creates an empty (all-zero-bytes) *heap file* where pm-`gawk` will store script variables; its size is a multiple of the system page size (4 KiB). Next, `export` sets an environment variable that enables pm-`gawk` to find the heap file; if `gawk` does *not* see this envar, persistence is not activated. The third command runs a one-line AWK script that initializes variable `myvar`, which will reside in the heap file and thereby outlive the interpreter process that initialized it. Finally, the fourth command invokes pm-`gawk` on a *different* one-line script that increments and prints `myvar`. The output shows that pm-`gawk` has indeed "remembered" `myvar` across executions of unrelated scripts. (If the `gawk` executable in your search `$PATH` lacks the persistence feature, the output in the above example will be '7' instead of '54'. See Appendix A [Installation], page 15.) To disable persistence until you want it again, prevent `gawk` from finding the heap file via `unset GAWK_PERSIST_FILE`. To permanently "forget" script variables, delete the heap file.

Toggling persistence by `export`-ing and `unset`-ing "ambient" envars requires care: Forgetting to `unset` when you no longer want persistence can cause confusing bugs. Fortunately, `bash` allows you to pass envars more deliberately, on a per-command basis:

```
$ rm heap.pma                       # start fresh
$ unset GAWK_PERSIST_FILE           # eliminate ambient envar
$ truncate -s 4096000 heap.pma      # create new heap file

$ GAWK_PERSIST_FILE=heap.pma gawk 'BEGIN{myvar = 47}'
$ gawk 'BEGIN{myvar += 7; print myvar}'
7
$ GAWK_PERSIST_FILE=heap.pma gawk 'BEGIN{myvar += 7; print myvar}'
54
```

The first `gawk` invocation sees the special envar prepended on the command line, so it activates pm-`gawk`. The second `gawk` invocation, however, does *not* see the envar and therefore does not access the script variable stored in the heap file. The third `gawk` invocation does see the special envar and therefore uses the script variable from the heap file.

While sometimes less error prone than ambient envars, per-command envar passing as shown above is verbose and shouty. A shell alias saves keystrokes and reduces visual clutter:

```
$ alias pm='GAWK_PERSIST_FILE=heap.pma'
$ pm gawk 'BEGIN{print ++myvar}'
55
$ pm gawk 'BEGIN{print ++myvar}'
56
```

3 Examples

Our first example uses pm-`gawk` to streamline analysis of a prose corpus, Mark Twain's *Tom Sawyer* and *Huckleberry Finn* from `https://gutenberg.org/files/74/74-0.txt` and `https://gutenberg.org/files/76/76-0.txt`. We first convert non-alphabetic characters to newlines (so each line has at most one word) and convert to lowercase:

```
$ tr -c a-zA-Z '\n' < 74-0.txt | tr A-Z a-z > sawyer.txt
$ tr -c a-zA-Z '\n' < 76-0.txt | tr A-Z a-z > finn.txt
```

It's easy to count word frequencies with AWK's associative arrays. pm-`gawk` makes these arrays persistent, so we need not re-ingest the entire corpus every time we ask a new question ("read once, analyze happily ever after"):

```
$ truncate -s 100M twain.pma
$ export GAWK_PERSIST_FILE=twain.pma
$ gawk '{ts[$1]++}' sawyer.txt                    # ingest
$ gawk 'BEGIN{print ts["work"], ts["play"]}'      # query
92 11
$ gawk 'BEGIN{print ts["necktie"], ts["knife"]}'  # query
2 27
```

The `truncate` command above creates a heap file large enough to store all of the data it must eventually contain, with plenty of room to spare. (As we'll see in Section 4.3 [Sparse Heap Files], page 8, this isn't wasteful). The `export` command ensures that subsequent `gawk` invocations activate pm-`gawk`. The first pm-`gawk` command stores *Tom Sawyer*'s word frequencies in associative array `ts[]`. Because this array is persistent, subsequent pm-`gawk` commands can access it without having to parse the input file again.

Expanding our analysis to encompass a second book is easy. Let's populate a new associative array `hf[]` with the word frequencies in *Huckleberry Finn*:

```
$ gawk '{hf[$1]++}' finn.txt
```

Now we can freely intermix accesses to both books' data conveniently and efficiently, without the overhead and coding fuss of repeated input parsing:

```
$ gawk 'BEGIN{print ts["river"], hf["river"]}'
26 142
```

By making AWK more interactive, pm-`gawk` invites casual conversations with data. If we're curious what words in *Finn* are absent from *Sawyer*, answers (including "flapdoodle," "yellocution," and "sockdolager") are easy to find:

```
$ gawk 'BEGIN{for(w in hf) if (!(w in ts)) print w}'
```

Rumors of Twain's death may be exaggerated. If he publishes new books in the future, it will be easy to incorporate them into our analysis incrementally. The performance benefits of incremental processing for common AWK chores such as log file analysis are discussed in `https://queue.acm.org/detail.cfm?id=3534855` and the companion paper cited therein, and below in Chapter 4 [Performance], page 6.

Exercise: The "Markov" AWK script on page 79 of Kernighan & Pike's *The Practice of Programming* generates random text reminiscent of a given corpus using a simple statistical modeling technique. This script consists of a "learning" or "training" phase followed by an output-generation phase. Use pm-`gawk` to de-couple the two phases and to allow the statistical model to incrementally ingest additions to the input corpus.

Our second example considers another domain that plays to AWK's strengths, data analysis. For simplicity we'll create two small input files of numeric data.

```
$ printf '1\n2\n3\n4\n5\n' > A.dat
$ printf '5\n6\n7\n8\n9\n' > B.dat
```

A conventional *non*-persistent AWK script can compute basic summary statistics:

```
$ cat summary_conventional.awk
    1 == NR  { min = max = $1 }
    min > $1 { min  = $1 }
    max < $1 { max  = $1 }
             { sum += $1 }
    END { print "min: " min " max: " max " mean: " sum/NR }

$ gawk -f summary_conventional.awk A.dat B.dat
min: 1 max: 9 mean: 5
```

To use pm-`gawk` for the same purpose, we first create a heap file for our AWK script variables and tell pm-`gawk` where to find it via the usual environment variable:

```
$ truncate -s 10M stats.pma
$ export GAWK_PERSIST_FILE=stats.pma
```

pm-`gawk` requires changing the above script to ensure that `min` and `max` are initialized exactly once, when the heap file is first used, and *not* every time the script runs. Furthermore, whereas script-defined variables such as `min` retain their values across pm-`gawk` executions, built-in AWK variables such as `NR` are reset to zero every time pm-`gawk` runs, so we can't use them in the same way. Here's a modified script for pm-`gawk`:

```
$ cat summary_persistent.awk
    ! init   { min = max = $1; init = 1 }
    min > $1 { min = $1 }
    max < $1 { max = $1 }
             { sum += $1; ++n }
    END { print "min: " min " max: " max " mean: " sum/n }
```

Note the different pattern on the first line and the introduction of `n` to supplant `NR`. When used with pm-`gawk`, this new initialization logic supports the same kind of cumulative processing that we saw in the text-analysis scenario. For example, we can ingest our input files separately:

```
$ gawk -f summary_persistent.awk A.dat
min: 1 max: 5 mean: 3

$ gawk -f summary_persistent.awk B.dat
min: 1 max: 9 mean: 5
```

As expected, after the second pm-`gawk` invocation consumes the second input file, the output matches that of the non-persistent script that read both files at once.

Exercise: Amend the AWK scripts above to compute the median and mode(s) using both conventional `gawk` and pm-`gawk`. (The median is the number in the middle of a sorted list; if the length of the list is even, average the two numbers at the middle. The modes are the values that occur most frequently.)

Our third and final set of examples shows that pm-gawk allows us to bundle both script-defined data and also user-defined *functions* in a persistent heap that may be passed freely between unrelated AWK scripts.

The following shell transcript repeatedly invokes pm-gawk to create and then employ a user-defined function. These separate invocations involve several different AWK scripts that communicate via the heap file. Each invocation can add user-defined functions and add or remove data from the heap that subsequent invocations will access.

```
$ truncate -s 10M funcs.pma
$ export GAWK_PERSIST_FILE=funcs.pma
$ gawk 'function count(A,t) {for(i in A)t++; return ""==t?0:t}'
$ gawk 'BEGIN { a["x"] = 4; a["y"] = 5; a["z"] = 6 }'
$ gawk 'BEGIN { print count(a) }'
3
$ gawk 'BEGIN { delete a["x"] }'
$ gawk 'BEGIN { print count(a) }'
2
$ gawk 'BEGIN { delete a }'
$ gawk 'BEGIN { print count(a) }'
0
$ gawk 'BEGIN { for (i=0; i<47; i++) a[i]=i }'
$ gawk 'BEGIN { print count(a) }'
47
```

The first pm-gawk command creates user-defined function `count()`, which returns the number of entries in a given associative array; note that variable `t` is local to `count()`, not global. The next pm-gawk command populates a persistent associative array with three entries; not surprisingly, the `count()` call in the following pm-gawk command finds these three entries. The next two pm-gawk commands respectively delete an array entry and print the reduced count, 2. The two commands after that delete the entire array and print a count of zero. Finally, the last two pm-gawk commands populate the array with 47 entries and count them.

The following shell script invokes pm-gawk repeatedly to create a collection of user-defined functions that perform basic operations on quadratic polynomials: evaluation at a given point, computing the discriminant, and using the quadratic formula to find the roots. It then factorizes $x^2 + x - 12$ into $(x - 3)(x + 4)$.

```
#!/bin/sh
rm -f                    poly.pma
truncate -s 10M          poly.pma
export GAWK_PERSIST_FILE=poly.pma
gawk 'function q(x) { return a*x^2 + b*x + c }'
gawk 'function p(x) { return "q(" x ") = " q(x) }'
gawk 'BEGIN { print p(2) }'           # evaluate & print
gawk 'BEGIN{ a = 1; b = 1; c = -12 }' # new coefficients
gawk 'BEGIN { print p(2) }'           # eval/print again
gawk 'function d(s) { return s * sqrt(b^2 - 4*a*c)}'
gawk 'BEGIN{ print "discriminant (must be >=0): " d(1)}'
gawk 'function r(s) { return (-b + d(s))/(2*a)}'
gawk 'BEGIN{ print "root: " r( 1) "    " p(r( 1)) }'
gawk 'BEGIN{ print "root: " r(-1) "    " p(r(-1)) }'
gawk 'function abs(n) { return n >= 0 ? n : -n }'
gawk 'function sgn(x) { return x >= 0 ? "- " : "+ " } '
gawk 'function f(s) { return "(x " sgn(r(s)) abs(r(s))}'
gawk 'BEGIN{ print "factor: " f( 1) ")" }'
gawk 'BEGIN{ print "factor: " f(-1) ")" }'
rm -f poly.pma
```

4 Performance

This chapter explains several performance advantages that result from the implementation of persistent memory in pm-`gawk`, shows how tuning the underlying operating system sometimes improves performance, and presents experimental performance measurements. To make the discussion concrete, we use examples from a GNU/Linux system—GNU utilities atop the Linux OS—but the principles apply to other modern operating systems.

4.1 Constant-Time Array Access

pm-`gawk` preserves the efficiency of data access when data structures are created by one process and later re-used by a different process.

Consider the associative arrays used to analyze Mark Twain's books in Chapter 3 [Examples], page 3. We created arrays `ts[]` and `hf[]` by reading files `sawyer.txt` and `finn.txt`. If N denotes the total volume of data in these files, building the associative arrays typically requires time proportional to N, or "*O(N)* expected time" in the lingo of asymptotic analysis. If W is the number of unique words in the input files, the size of the associative arrays will be proportional to W, or *O(W)*. Accessing individual array elements requires only *constant* or *O(1)* expected time, not *O(N)* or *O(W)* time, because `gawk` implements arrays as hash tables.

The performance advantage of pm-`gawk` arises when different processes create and access associative arrays. Accessing an element of a persistent array created by a previous pm-`gawk` process, as we did earlier in `BEGIN{print ts["river"], hf["river"]}`, still requires only *O(1)* time, which is asymptotically far superior to the alternatives. Naïvely reconstructing arrays by re-ingesting all raw inputs in every `gawk` process that accesses the arrays would of course require *O(N)* time—a profligate cost if the text corpus is large. Dumping arrays to files and re-loading them as needed would reduce the preparation time for access to *O(W)*. That can be a substantial improvement in practice; N is roughly 19 times larger than W in our Twain corpus. Nonetheless *O(W)* remains far slower than pm-`gawk`'s *O(1)*. As we'll see in Section 4.6 [Results], page 13, the difference is not merely theoretical.

The persistent memory implementation beneath pm-`gawk` enables it to avoid work proportional to N or W when accessing an element of a persistent associative array. Under the hood, pm-`gawk` stores script-defined AWK variables such as associative arrays in a persistent heap laid out in a memory-mapped file (the heap file). When an AWK script accesses an element of an associative array, pm-`gawk` performs a lookup on the corresponding hash table, which in turn accesses memory on the persistent heap. Modern operating systems implement memory-mapped files in such a way that these memory accesses trigger the bare minimum of data movement required: Only those parts of the heap file containing needed data are "paged in" to the memory of the pm-`gawk` process. In the worst case, the heap file is not in the file system's in-memory cache, so the required pages must be faulted into memory from storage. Our asymptotic analysis of efficiency applies regardless of whether the heap file is cached or not. The entire heap file is *not* accessed merely to access an element of a persistent associative array.

Persistent memory thus enables pm-`gawk` to offer the flexibility of de-coupling data ingestion from analytic queries without the fuss and overhead of serializing and loading data structures and without sacrificing constant-time access to the associative arrays that make AWK scripting convenient and productive.

4.2 Virtual Memory and Big Data

Small data sets seldom spoil the delights of AWK by causing performance troubles, with or without persistence. As the size of the `gawk` interpreter's internal data structures approaches the capacity of physical memory, however, acceptable performance requires understanding modern operating systems and sometimes tuning them. Fortunately pm-`gawk` offers new degrees of control for performance-conscious users tackling large data sets. A terse mnemonic captures the basic principle: Precluding paging promotes peak performance and prevents perplexity.

Modern operating systems feature *virtual memory* that strives to appear both larger than installed DRAM (which is small) and faster than installed storage devices (which are slow). As a program's memory footprint approaches the capacity of DRAM, the virtual memory system transparently *pages* (moves) the program's data between DRAM and a *swap area* on a storage device. Paging can degrade performance mildly or severely, depending on the program's memory access patterns. Random accesses to large data structures can trigger excessive paging and dramatic slowdown. Unfortunately, the hash tables beneath AWK's signature associative arrays inherently require random memory accesses, so large associative arrays can be problematic.

Persistence changes the rules in our favor: The OS pages data to pm-`gawk`'s *heap file* instead of the swap area. This won't help performance much if the heap file resides in a conventional storage-backed file system. On Unix-like systems, however, we may place the heap file in a DRAM-backed file system such as `/dev/shm/`, which entirely prevents paging to slow storage devices. Temporarily placing the heap file in such a file system is a reasonable expedient, with two caveats: First, keep in mind that DRAM-backed file systems perish when the machine reboots or crashes, so you must copy the heap file to a conventional storage-backed file system when your computation is done. Second, pm-`gawk`'s memory footprint can't exceed available DRAM if you place the heap file in a DRAM-backed file system.

Tuning OS paging parameters may improve performance if you decide to run pm-`gawk` with a heap file in a conventional storage-backed file system. Some OSes have unhelpful default habits regarding modified ("dirty") memory backed by files. For example, the OS may write dirty memory pages to the heap file periodically and/or when the OS believes that "too much" memory is dirty. Such "eager" writeback can degrade performance noticeably and brings no benefit to pm-`gawk`. Fortunately some OSes allow paging defaults to be over-ridden so that writeback is "lazy" rather than eager. For Linux see the discussion of the `dirty_*` parameters at https://www.kernel.org/doc/html/latest/admin-guide/sysctl/vm.html. Changing these parameters can prevent wasteful eager paging:[1]

```
$ echo 100    | sudo tee /proc/sys/vm/dirty_background_ratio
$ echo 100    | sudo tee /proc/sys/vm/dirty_ratio
$ echo 300000 | sudo tee /proc/sys/vm/dirty_expire_centisecs
$ echo 50000  | sudo tee /proc/sys/vm/dirty_writeback_centisecs
```

Tuning paging parameters can help non-persistent `gawk` as well as pm-`gawk`. [Disclaimer: OS tuning is an occult art, and your mileage may vary.]

[1] The `tee` rigmarole is explained at https://askubuntu.com/questions/1098059/which-is-the-right-way-to-drop-caches-in-lubuntu.

4.3 Sparse Heap Files

To be frugal with storage resources, pm-`gawk`'s heap file should be created as a *sparse file*: a file whose logical size is larger than its storage resource footprint. Modern file systems support sparse files, which are easy to create using the `truncate` tool shown in our examples.

Let's first create a conventional *non*-sparse file using `echo`:

```
$ echo hi > dense
$ ls -l dense
-rw-rw-r--. 1 me me 3 Aug  5 23:08 dense
$ du -h dense
4.0K    dense
```

The `ls` utility reports that file `dense` is three bytes long (two for the letters in "hi" plus one for the newline). The `du` utility reports that this file consumes 4 KiB of storage—one block of disk, as small as a non-sparse file's storage footprint can be. Now let's use `truncate` to create a logically enormous sparse file and check its physical size:

```
$ truncate -s 1T sparse
$ ls -l sparse
-rw-rw-r--. 1 me me 1099511627776 Aug  5 22:33 sparse
$ du -h sparse
0       sparse
```

Whereas `ls` reports the logical file size that we expect (one TiB or 2 raised to the power 40 bytes), `du` reveals that the file occupies no storage whatsoever. The file system will allocate physical storage resources beneath this file as data is written to it; reading unwritten regions of the file yields zeros.

The "pay as you go" storage cost of sparse files offers both convenience and control for pm-`gawk` users. If your file system supports sparse files, go ahead and create lavishly capacious heap files for pm-`gawk`. Their logical size costs nothing and persistent memory allocation within pm-`gawk` won't fail until physical storage resources beneath the file system are exhausted. But if instead you want to *prevent* a heap file from consuming too much storage, simply set its initial size to whatever bound you wish to enforce; it won't eat more disk than that. Copying sparse files with GNU `cp` creates sparse copies by default.

File-system encryption can preclude sparse files: If the cleartext of a byte offset range within a file is all zero bytes, the corresponding ciphertext probably shouldn't be all zeros! Encrypting at the storage layer instead of the file system layer may offer acceptable security while still permitting file systems to implement sparse files.

Sometimes you might prefer a dense heap file backed by pre-allocated storage resources, for example to increase the likelihood that pm-`gawk`'s internal memory allocation will succeed until the persistent heap occupies the entire heap file. The `fallocate` utility will do the trick:

```
$ fallocate -l 1M mibi
$ ls -l mibi
-rw-rw-r--. 1 me me 1048576 Aug  5 23:18 mibi
$ du -h mibi
1.0M    mibi
```

We get the MiB we asked for, both logically and physically.

4.4 Persistence versus Durability

Arguably the most important general guideline for good performance in computer systems is, "pay only for what you need."[1] To apply this maxim to pm-gawk we must distinguish two concepts that are frequently conflated: persistence and durability.[2] (A third logically distinct concept is the subject of Chapter 5 [Data Integrity], page 14.)

Persistent data outlive the processes that access them, but don't necessarily last forever. For example, as explained in `man mq_overview`, message queues are persistent because they exist until the system shuts down. *Durable* data reside on a physical medium that retains its contents even without continuously supplied power. For example, hard disk drives and solid state drives store durable data. Confusion arises because persistence and durability are often correlated: Data in ordinary file systems backed by HDDs or SSDs are typically both persistent and durable. Familiarity with `fsync()` and `msync()` might lead us to believe that durability is a subset of persistence, but in fact the two characteristics are orthogonal: Data in the swap area are durable but not persistent; data in DRAM-backed file systems such as `/dev/shm/` are persistent but not durable.

Durability often costs more than persistence, so performance-conscious pm-gawk users pay the added premium for durability only when persistence alone is not sufficient. Two ways to avoid unwanted durability overheads were discussed in Section 4.2 [Virtual Memory and Big Data], page 7: Place pm-gawk's heap file in a DRAM-backed file system, or disable eager writeback to the heap file. Expedients such as these enable you to remove durability overheads from the critical path of multi-stage data analyses even when you want heap files to eventually be durable: Allow pm-gawk to run at full speed with persistence alone; force the heap file to durability (using the `cp` and `sync` utilities as necessary) after output has been emitted to the next stage of the analysis and the pm-gawk process using the heap has terminated.

Experimenting with synthetic data builds intuition for how persistence and durability affect performance. You can write a little AWK or C program to generate a stream of random text, or just cobble together a quick and dirty generator on the command line:

```
$ openssl rand --base64 1000000 | tr -c a-zA-Z '\n' > random.dat
```

Varying the size of random inputs can, for example, find where performance "falls off the cliff" as pm-gawk's memory footprint exceeds the capacity of DRAM and paging begins.

Experiments require careful methodology, especially when the heap file is in a storage-backed file system. Overlooking the file system's DRAM cache can easily misguide interpretation of results and foil repeatability. Fortunately Linux allows us to invalidate the file system cache and thus mimic a "cold start" condition resembling the immediate aftermath of a machine reboot. Accesses to ordinary files on durable storage will then be served from the storage devices, not from cache. Read about `sync` and `/proc/sys/vm/drop_caches` at `https://www.kernel.org/doc/html/latest/admin-guide/sysctl/vm.html`.

[1] Remarkably, this guideline is widely ignored in surprising ways. Certain well-known textbook algorithms continue to grind away fruitlessly long after having computed all of their output. See `https://queue.acm.org/detail.cfm?id=3424304`.

[2] In recent years the term "persistent memory" has sometimes been used to denote novel byte-addressable non-volatile memory hardware—an unfortunate practice that contradicts sensible long-standing convention and causes needless confusion. NVM provides durability. Persistent memory is a software abstraction that doesn't require NVM. See `https://queue.acm.org/detail.cfm?id=3358957`.

4.5 Experiments

The C-shell (`csh`) script listed below illustrates concepts and implements tips presented in this chapter. It produced the results discussed in Section 4.6 [Results], page 13, in roughly 20 minutes on an aging laptop. You can cut and paste the code listing below into a file, or download it from `http://web.eecs.umich.edu/~tpkelly/pma/`.

The script measures the performance of four different ways to support word frequency queries over a text corpus: The naïve approach of reading the corpus into an associative array for every query; manually dumping a text representation of the word-frequency table and manually loading it prior to a query; using `gawk`'s `rwarray` extension to dump and load an associative array; and using pm-`gawk` to maintain a persistent associative array.

Comments at the top explain prerequisites. Lines 8–10 set input parameters: the directory where tests are run and where files including the heap file are held, the off-the-shelf timer used to measure run times and other performance characteristics such as peak memory usage, and the size of the input. The default input size results in pm-`gawk` memory footprints under 3 GiB, which is large enough for interesting results and small enough to fit in DRAM and avoid paging on today's computers. Lines 13–14 define a homebrew timer.

Two sections of the script are relevant if the default run directory is changed from `/dev/shm/` to a directory in a conventional storage-backed file system: Lines 15–17 define the mechanism for clearing file data cached in DRAM; lines 23–30 set Linux kernel parameters to discourage eager paging.

Lines 37–70 spit out, compile, and run a little C program to generate a random text corpus. This program is fast, flexible, and deterministic, generating the same random output given the same parameters.

Lines 71–100 run the four different AWK approaches on the same random input, reporting separately the time to build and to query the associative array containing word frequencies.

```
#!/bin/csh -f
# Set PMG envar to path of pm-gawk executable and AWKLIBPATH                        #   2
#    to find rwarray.so                                                             #   3
# Requires "sudo" to work; consider this for /etc/sudoers file:                     #   4
#    Defaults:youruserid    !authenticate                                           #   5
echo ’begin:      ’ ‘date‘ ‘date +%s‘                                               #   6
unsetenv GAWK_PERSIST_FILE  # disable persistence until wanted                      #   7
set dir = ’/dev/shm’        # where heap file et al. will live                      #   8
set tmr = ’/usr/bin/time’   # can also use shell built-in "time"                    #   9
set isz = 1073741824        # input size; 1 GiB                                     #  10
# set isz = 100000000       # small input for quick testing                         #  11
cd $dir   # tick/tock/tyme below are homebrew timer, good within ~2ms               #  12
alias tick ’set t1 = ‘date +%s.%N‘’ ; alias tock ’set t2 = ‘date +%s.%N‘’           #  13
alias tyme ’$PMG -v t1=$t1 -v t2=$t2 "BEGIN{print t2-t1}"’                          #  14
alias tsync ’tick ; sync ; tock ; echo "sync time:  " ‘tyme‘’                       #  15
alias drop_caches ’echo 3 | sudo tee /proc/sys/vm/drop_caches’                      #  16
alias snd ’tsync; drop_caches’                                                      #  17
echo "pm-gawk:     $PMG" ; echo ’std gawk:   ’ ‘which gawk‘                         #  18
echo "run dir:     $dir" ; echo ’pwd:        ’ ‘pwd‘                                #  19
echo ’dir content:’      ; ls -l $dir |& $PMG ’{print "             " $0}’          #  20
echo ’timer:      ’ $tmr ; echo ’AWKLIBPATH: ’ $AWKLIBPATH                          #  21
echo 'OS params:' ; set vm = '/proc/sys/vm/dirty'
sudo sh -c "echo 100     > ${vm}_background_ratio"     # restore these
sudo sh -c "echo 100     > ${vm}_ratio"                # paging params
sudo sh -c "echo 1080000 > ${vm}_expire_centisecs"     # to defaults
sudo sh -c "echo 1080000 > ${vm}_writeback_centisecs"  # if necessary
foreach d ( ${vm}_background_ratio  ${vm}_ratio \
            ${vm}_expire_centisecs  ${vm}_writeback_centisecs )
    printf "             %-38s %7d\n" $d `cat $d`
end
tick ; tock ; echo 'timr ovrhd: ' `tyme` 's (around 2ms for TK)'
tick ; $PMG 'BEGIN{print "pm-gawk?     yes"}'
tock ; echo 'pmg ovrhd:  ' `tyme` 's (around 4-5 ms for TK)'
set inp = 'input.dat'
echo 'input size  ' $isz
echo "input file:  $inp"
set rg = rgen  # spit out and compile C program to generate random inputs
rm -f $inp  $rg.c $rg
cat <<EOF > $rg.c
// generate N random words, one per line, no blank lines
// charset is e.g. 'abcdefg@' where '@' becomes newline
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#define RCH     c = a[rand() % L];
#define PICK    do { RCH } while (0)
#define PICKCH  do { RCH } while (c == '@')
#define FP(...) fprintf(stderr, __VA_ARGS__)
int main(int argc, char *argv[]) {
  if (4 != argc) {
    FP("usage:  %s  charset  N  seed\n",
                    argv[0]);   return 1; }
  char c, *a =      argv[1];    size_t L = strlen(a);
  long int N = atol(argv[2]);
  srand(       atol(argv[3]));
  if (2 > N) { FP("N == %ld < 2\n", N); return 2; }
  PICKCH;
  for (;;) {
    if (2 == N) { PICKCH; putchar(c); putchar('\n'); break; }
    if ('@' == c) { putchar('\n'); PICKCH; }
    else          { putchar( c  ); PICK;   }
    if (0 >= --N) break;
  }
}
EOF
gcc -std=c11 -Wall -Wextra -O3 -o $rg $rg.c
set t = '@@@@@@@' ; set c = "abcdefghijklmnopqrstuvwxyz$t$t$t$t$t$t"
tick ; ./$rg "$c" $isz 47 > $inp ; tock ; echo 'gen time:   ' `tyme`
echo "input file:  $inp"
echo 'input wc:   ' `wc < $inp` ; echo 'input uniq: ' `sort -u $inp | wc`
```

```
snd ############################################################################    #  71
tick ; $tmr $PMG ’{n[$1]++}END{print "output: " n["foo"]}’ $inp                     #  72
tock ; echo ’T naive O(N):          ’ ‘tyme‘ ; echo ’’                             #  73
rm -f rwa                                                                           #  74
snd ############################################################################    #  75
echo ’’                                                                             #  76
tick ; $tmr $PMG -l rwarray ’{n[$1]++}END{print "writea",writea("rwa",n)}’ $inp     #  77
tock ; echo ’T rwarray build O(N):  ’ ‘tyme‘ ; echo ’’                             #  78
snd  # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # #    #  79
tick ; $tmr $PMG -l rwarray ’BEGIN{print "reada",reada("rwa",n); \                  #  80
                                   print "output: " n["foo"]}’                      #  81
tock ; echo ’T rwarray query O(W):   ’ ‘tyme‘ ; echo ’’                            #  82
rm -f ft                                                                            #  83
snd ############################################################################    #  84
tick ; $tmr $PMG ’{n[$1]++}END{for(w in n)print n[w], w}’ $inp > ft                 #  85
tock ; echo ’T freqtbl build O(N):  ’ ‘tyme‘ ; echo ’’                             #  86
snd  # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # #    #  87
tick ; $tmr $PMG ’{n[$2] = $1}END{print "output: " n["foo"]}’ ft                    #  88
tock ; echo ’T freqtbl query O(W):   ’ ‘tyme‘ ; echo ’’                            #  89
rm -f heap.pma                                                                      #  90
snd ############################################################################    #  91
truncate -s 3G heap.pma  # enlarge if needed                                        #  92
setenv GAWK_PERSIST_FILE heap.pma                                                   #  93
tick ; $tmr $PMG ’{n[$1]++}’ $inp                                                   #  94
tock ; echo ’T pm-gawk build O(N):  ’ ‘tyme‘ ; echo ’’                             #  95
snd  # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # #    #  96
tick ; $tmr $PMG ’BEGIN{print "output: " n["foo"]}’                                 #  97
tock ; echo ’T pm-gawk query O(1):    ’ ‘tyme‘ ; echo ’’                           #  98
unsetenv GAWK_PERSIST_FILE                                                          #  99
snd ############################################################################    # 100
echo ’Note:  all output lines above should be identical’ ; echo ’’                  # 101
echo ’dir content:’ ; ls -l $dir |& $PMG ’{print "             " $0}’               # 102
echo ’’ ; echo ’storage footprints:’                                                # 103
foreach f ( rwa ft heap.pma )  # compression is very slow, so we comment it out     # 104
    echo "    $f " ‘du -BK $dir/$f‘ # ‘xz --best < $dir/$f | wc -c‘ ’bytes xz’      # 105
end                                                                                 # 106
echo ’’ ; echo ’end:        ’ ‘date‘ ‘date +%s‘ ; echo ’’                           # 107
```

4.6 Results

Running the script of Section 4.5 [Experiments], page 10, with default parameters on an aging laptop yielded the results summarized in the table below. More extensive experiments, not reported here, yield qualitatively similar results. Keep in mind that performance measurements are often sensitive to seemingly irrelevant factors. For example, the program that runs first may have the advantage of a cooler CPU; later contestants may start with a hot CPU and consequent clock throttling by a modern processor's thermal regulation apparatus. Very generally, performance measurement is a notoriously tricky business. For scripting, whose main motive is convenience rather than speed, the proper role for performance measurements is to qualitatively test hypotheses such as those that follow from asymptotic analyses and to provide a rough idea of when various approaches are practical.

AWK script		run time (sec)	peak memory footprint (K)	intermediate storage (K)
naive	O(N)	242.132	2,081,360	n/a
rwarray build	O(N)	250.288	2,846,868	156,832
rwarray query	O(W)	11.653	2,081,444	
freqtbl build	O(N)	288.408	2,400,120	69,112
freqtbl query	O(W)	11.624	2,336,616	
pm-gawk build	O(N)	251.946	2,079,520	2,076,608
pm-gawk query	O(1)	0.026	3,252	

The results are consistent with the asymptotic analysis of Section 4.1 [Constant-Time Array Access], page 6. All four approaches require roughly four minutes to read the synthetic input data. The naïve approach must do this every time it performs a query, but the other three build an associative array to support queries and separately serve such queries. The `freqtbl` and `rwarray` approaches build an associative array of word frequencies, serialize it to an intermediate file, and then read the entire intermediate file prior to serving queries; the former does this manually and the latter uses a `gawk` extension. Both can serve queries in roughly ten seconds, not four minutes. As we'd expect from the asymptotic analysis, performing work proportional to the number of words is preferable to work proportional to the size of the raw input corpus: *O(W)* time is faster than *O(N)*. And as we'd expect, pm-`gawk`'s constant-time queries are faster still, by roughly two orders of magnitude. For the computations considered here, pm-`gawk` makes the difference between blink-of-an-eye interactive queries and response times long enough for the user's mind to wander.

Whereas `freqtbl` and `rwarray` reconstruct an associative array prior to accessing an individual element, pm-`gawk` stores a ready-made associative array in persistent memory. That's why its intermediate file (the heap file) is much larger than the other two intermediate files, why the heap file is nearly as large as pm-`gawk`'s peak memory footprint while building the persistent array, and why its memory footprint is very small while serving a query that accesses a single array element. The upside of the large heap file is *O(1)* access instead of *O(W)*—a classic time-space tradeoff. If storage is a scarce resource, all three intermediate files can be compressed, `freqtbl` by a factor of roughly 2.7, `rwarray` by roughly 5.6x, and pm-`gawk` by roughly 11x using `xz`. Compression is CPU-intensive and slow, another time-space tradeoff.

5 Data Integrity

Mishaps including power outages, OS kernel panics, scripting bugs, and command-line typos can harm your data, but precautions can mitigate these risks. In scripting scenarios it usually suffices to create safe backups of important files at appropriate times. As simple as this sounds, care is needed to achieve genuine protection and to reduce the costs of backups. Here's a prudent yet frugal way to back up a heap file between uses:

```
$ backup_base=heap_bk_‘date +%s‘
$ cp --reflink=always heap.pma $backup_base.pma
$ chmod a-w $backup_base.pma
$ sync
$ touch $backup_base.done
$ chmod a-w $backup_base.done
$ sync
$ ls -l heap*
-rw-rw-r--. 1 me me 4096000 Aug  6 15:53 heap.pma
-r--r--r--. 1 me me       0 Aug  6 16:16 heap_bk_1659827771.done
-r--r--r--. 1 me me 4096000 Aug  6 16:16 heap_bk_1659827771.pma
```

Timestamps in backup filenames make it easy to find the most recent copy if the heap file is damaged, even if last-mod metadata are inadvertently altered.

The `cp` command's `--reflink` option reduces both the storage footprint of the copy and the time required to make it. Just as sparse files provide "pay as you go" storage footprints, reflink copying offers "pay as you *change*" storage costs.[1] A reflink copy shares storage with the original file. The file system ensures that subsequent changes to either file don't affect the other. Reflink copying is not available on all file systems; XFS, BtrFS, and OCFS2 currently support it.[2] Fortunately you can install, say, an XFS file system *inside an ordinary file* on some other file system, such as `ext4`.[3]

After creating a backup copy of the heap file we use `sync` to force it down to durable media. Otherwise the copy may reside only in volatile DRAM memory—the file system's cache—where an OS crash or power failure could corrupt it.[4] After `sync`-ing the backup we create and `sync` a "success indicator" file with extension `.done` to address a nasty corner case: Power may fail *while* a backup is being copied from the primary heap file, leaving either file, or both, corrupt on storage—a particularly worrisome possibility for jobs that run unattended. Upon reboot, each `.done` file attests that the corresponding backup succeeded, making it easy to identify the most recent successful backup.

Finally, if you're serious about tolerating failures you must "train as you would fight" by testing your hardware/software stack against realistic failures. For realistic power-failure testing, see https://queue.acm.org/detail.cfm?id=3400902.

[1] The system call that implements reflink copying is described in `man ioctl_ficlone`.

[2] The `--reflink` option creates copies as sparse as the original. If reflink copying is not available, `--sparse=always` should be used.

[3] See https://www.usenix.org/system/files/login/articles/login_winter19_08_kelly.pdf.

[4] On some OSes `sync` provides very weak guarantees, but on Linux `sync` returns only after all file system data are flushed down to durable storage. If your `sync` is unreliable, write a little C program that calls `fsync()` to flush a file. To be safe, also call `fsync()` on every enclosing directory on the file's `realpath()` up to the root.

6 Acknowledgments

Haris Volos, Zi Fan Tan, and Jianan Li developed a persistent `gawk` prototype based on a fork of the `gawk` source. Advice from `gawk` maintainer Arnold Robbins to me, which I forwarded to them, proved very helpful. Robbins moreover implemented, documented, and tested pm-`gawk` for the official version of `gawk`; along the way he suggested numerous improvements for the `pma` memory allocator beneath pm-`gawk`. Corinna Vinschen suggested other improvements to `pma` and tested pm-`gawk` on Cygwin. Nelson H. F. Beebe provided access to Solaris machines for testing. Robbins, Volos, Li, Tan, Jon Bentley, and Hans Boehm reviewed drafts of this user manual and provided useful feedback. Bentley suggested the min/max/mean example in Chapter 3 [Examples], page 3, and also the exercise of making Kernighan & Pike's "Markov" script persistent. Volos provided and tested the advice on tuning OS parameters in Section 4.2 [Virtual Memory and Big Data], page 7. Stan Park provided insights about virtual memory, file systems, and utilities.

Appendix A Installation

`gawk` 5.2 featuring persistent memory is expected to be released in September 2022; look for it at `http://ftp.gnu.org/gnu/gawk/`. If 5.2 is not released yet, the master git branch is available at `http://git.savannah.gnu.org/cgit/gawk.git/snapshot/gawk-master.tar.gz`. Unpack the tarball, run `./bootstrap.sh`, `./configure`, `make`, and `make check`, then try some of the examples presented earlier. In the normal course of events, 5.2 and later `gawk` releases featuring pm-`gawk` will appear in the software package management systems of major GNU/Linux distros. Eventually pm-`gawk` will be available in the default `gawk` on such systems.

Appendix B Debugging

For bugs unrelated to persistence, see the `gawk` documentation, e.g., *GAWK: Effective AWK Programming*, available at `https://www.gnu.org/software/gawk/manual/`.

If pm-`gawk` doesn't behave as you expect, first consider whether you're using the heap file that you intend; using the wrong heap file is a common mistake. Other fertile sources of bugs for newcomers are the fact that a `BEGIN` block is executed every time pm-`gawk` runs, which isn't always what you really want, and the fact that built-in AWK variables such as `NR` are always reset to zero every time the interpreter runs. See the discussion of initialization surrounding the min/max/mean script in Chapter 3 [Examples], page 3.

If you suspect a persistence-related bug in pm-`gawk`, you can set an environment variable that will cause its persistent heap module, `pma`, to emit more verbose error messages; for details see the main `gawk` documentation.

Programmers: You can re-compile `gawk` with assertions enabled, which will trigger extensive integrity checks within `pma`. Ensure that `pma.c` is compiled *without* the `-DNDEBUG` flag when `make` builds `gawk`. Run the resulting executable on small inputs, because the integrity checks can be very slow. If assertions fail, that likely indicates bugs somewhere in pm-`gawk`. Report such bugs to me (Terence Kelly) and also following the procedures in the main `gawk` documentation. Specify what version of `gawk` you're using, and try to provide a small and simple script that reliably reproduces the bug.

Appendix C History

The pm-`gawk` persistence feature is based on a new persistent memory allocator, `pma`, whose design is described in `https://queue.acm.org/detail.cfm?id=3534855`. It is instructive to trace the evolutionary paths that led to `pma` and pm-`gawk`.

I wrote many AWK scripts during my dissertation research on Web caching twenty years ago, most of which processed log files from Web servers and Web caches. Persistent `gawk` would have made these scripts smaller, faster, and easier to write, but at the time I was unable even to imagine that pm-`gawk` is possible. So I wrote a lot of bothersome, inefficient code that manually dumped and re-loaded AWK script variables to and from text files. A decade would pass before my colleagues and I began to connect the dots that make persistent scripting possible, and a further decade would pass before pm-`gawk` came together.

Circa 2011 while working at HP Labs I developed a fault-tolerant distributed computing platform called "Ken," which contained a persistent memory allocator that resembles a simplified `pma`: It presented a `malloc()`-like C interface and it allocated memory from a file-backed memory mapping. Experience with Ken convinced me that the software abstraction of persistent memory offers important attractions compared with the alternatives for managing persistent data (e.g., relational databases and key-value stores). Unfortunately, Ken's allocator is so deeply intertwined with the rest of Ken that it's essentially inseparable; to enjoy the benefits of Ken's persistent memory, one must "buy in" to a larger and more complicated value proposition. Whatever its other virtues might be, Ken isn't ideal for showcasing the benefits of persistent memory in isolation.

Another entangled aspect of Ken was a crash-tolerance mechanism that, in retrospect, can be viewed as a user-space implementation of failure-atomic `msync()`. The first post-Ken disentanglement effort isolated the crash-tolerance mechanism and implemented it in the Linux kernel, calling the result "failure-atomic `msync()`" (FAMS). FAMS strengthens the semantics of ordinary standard `msync()` by guaranteeing that the durable state of a memory-mapped file always reflects the most recent successful `msync()` call, even in the presence of failures such as power outages and OS or application crashes. The original Linux kernel FAMS prototype is described in a paper by Park et al. in EuroSys 2013. My colleagues and I subsequently implemented FAMS in several different ways including in file systems (FAST 2015) and user-space libraries. My most recent FAMS implementation, which leverages the reflink copying feature described elsewhere in this manual, is now the foundation of a new crash-tolerance feature in the venerable and ubiquitous GNU `dbm` (`gdbm`) database (`https://queue.acm.org/detail.cfm?id=3487353`).

In recent years my attention has returned to the advantages of persistent memory programming, lately a hot topic thanks to the commercial availability of byte-addressable non-volatile memory hardware (which, confusingly, is nowadays marketed as "persistent memory"). The software abstraction of persistent memory and the corresponding programming style, however, are perfectly compatible with *conventional* computers—machines with neither non-volatile memory nor any other special hardware or software. I wrote a few papers making this point, for example `https://queue.acm.org/detail.cfm?id=3358957`.

In early 2022 I wrote a new stand-alone persistent memory allocator, `pma`, to make persistent memory programming easy on conventional hardware. The `pma` interface is compatible with `malloc()` and, unlike Ken's allocator, `pma` is not coupled to a particular crash-tolerance mechanism. Using `pma` is easy and, at least to some, enjoyable.

Ken had been integrated into prototype forks of both the V8 JavaScript interpreter and a Scheme interpreter, so it was natural to consider whether `pma` might similarly enhance an interpreted scripting language. GNU AWK was a natural choice because the source code is orderly and because `gawk` has a single primary maintainer with an open mind regarding new features.

Jianan Li, Zi Fan Tan, Haris Volos, and I began considering persistence for `gawk` in late 2021. While I was writing `pma`, they prototyped pm-`gawk` in a fork of the `gawk` source. Experience with the prototype confirmed the expected convenience and efficiency benefits of pm-`gawk`, and by spring 2022 Arnold Robbins was implementing persistence in the official version of `gawk`. The persistence feature in official `gawk` differs slightly from the prototype: The former uses an environment variable to pass the heap file name to the interpreter whereas the latter uses a mandatory command-line option. In many respects, however, the two implementations are similar. A description of the prototype, including performance measurements, is available at `http://nvmw.ucsd.edu/nvmw2022-program/nvmw2022-data/nvmw2022-paper35-final_version_your_extended_abstract.pdf`.

I enjoy several aspects of pm-`gawk`. It's unobtrusive; as you gain familiarity and experience, it fades into the background of your scripting. It's simple in both concept and implementation, and more importantly it simplifies your scripts; much of its value is measured not in the code it enables you to write but rather in the code it lets you discard. It's all that I needed for my dissertation research twenty years ago, and more. Anecdotally, it appears to inspire creativity in early adopters, who have devised uses that pm-`gawk`'s designers never anticipated. I'm curious to see what new purposes you find for it.