Don’t MAWK AWK – the fastest and most elegant big data munging language!

Posted on September 10, 2009

update 2012-10-25: I’ve been informed there is a new maintainer for Mawk, who has probably fixed the bugs I’ve been seeing.

> From: Gert Hulselmans
>
> [The bugs you have found are] indeed true with mawk v1.3.3 which comes standard with Debian/Ubuntu. This version is almost not developed the last 10 years.
>
> I now already use mawk v1.3.4 maintained by another developer (Thomas E. Dickey)
> for more than a year on huge datafiles (sometimes several GB).
>
> The problems/wrong results I had with mawk v1.3.3 sometimes are gone. In his version, normally all open/known bugs are fixed.
>
> This version can be downloaded from: http://invisible-island.net/mawk/

update 2010-04-30: I have since found large datasets where mawk is buggy and gives the wrong result. nawk seems safe.

When one of these newfangled “Big Data” sets comes your way, the very first thing you have to do is data munging: shuffling around file formats, renaming fields and the like. Once you’re dealing with hundreds of megabytes of data, even simple operations can take plenty of time.

For one recent ad-hoc task I had — reformatting 1GB of textual feature data into a form Matlab and R can read — I tried writing implementations in several languages, with help from my classmate Elijah. The results really surprised us:

Language	Time (min:sec)	Speed (vs. gawk)	Lines of code	Notes	Type
mawk	1:06	7.8x	3	Mike Brennan’s Awk, system default on Ubuntu/Debian Linux.	VM
java	1:20	6.4x	32	version 1.6 (-server didn’t matter)	VM+JIT
c-ish c++	1:35	5.4x	42	g++ 4.0.1 with -O3, using stdio.h	Native
python	2:15	3.8x	20	version 2.5, system default on OSX 10.5	VM
perl	3:00	2.9x	17	version 5.8.8, system default on OSX 10.5	VM
nawk	6:10	1.4x	3	Brian Kernighan‘s “One True Awk”, system default on OSX, *BSD	?
c++	6:50	1.3x	48	g++ 4.0.1 with -O3, using fstream, stringstream	Native
ruby	7:30	1.1x	22	version 1.8.4, system default on OSX 10.5; also tried 1.9, but was slower	Interpreted
gawk	8:35	1x	3	GNU Awk, system default on RedHat/Fedora Linux	Interpreted

To be clear, the problem is to take several files of (item name, feature name, value) triples, like:

```
000794107-10-K-19960401 limited 1
000794107-10-K-19960401 colleges 1
```

```
000794107-10-K-19960401 code 2
...
004334108-10-K-19961230 recognition 1
004334108-10-K-19961230 gross 8
...
```

And then rename items and features into sequential numbers as a sparse matrix: (i, j, value) triples. Items should count up from inside each file; but features should be shared across files, so they need a shared counter. Finally, we need to write a mapping of feature IDs back to their names for later inspection; this can just be a list.

This task is simple, but it's representative of many data munging tasks out there. It inputs and outputs textual data. It's probably one-off. The algorithm is easy — especially since it's a subtask of something larger — but still complex enough you'll need a debug cycle or two. You want to get it done fast so you can get on to the real work. Complex programming tools, like debuggers, are of little use — you figure out what's going on by inspecting the output. Complex data processing environments, like Hadoop or an RDBMS, are also of little use — you have to munge in the first place to load data into them.

It turns out, this is a task AWK was made for. It's a language dating from the original Bell Labs Unix era — circa 1977 — and it's extremely specialized for processing delimited text files in a single pass. Perl was created in part to supersede it, but for this core use case, Awk is much more elegant and clearer. The implementation here is only 8 lines of code, expanded from merely 3 when I first wrote it.

Since it's a standardized language, many implementations exist. One of them, MAWK, is incredibly efficient. It outperforms ***all*** other languages, including statically typed compiled ones like Java and C++! It wins on *both* LOC and performance criteria — a rare feat indeed, transcending the usual competition of slow-but-easy scripting languages versus fast-but-hard compiled languages.

[There's another big pro-VM story here: Java beat C++, both in LOC/ease-of-programming as well as performance. C++ was, as usual, a total nightmare to write. What you don't see in the LOC numbers is the sheer amount of time spent googling through every weird issue. For example, apparently g++ requires you to define the hash function in order to use a hash_map of string keys. Or, there are a zillion different ways to split a string, none of them standard. Then there are 2 different I/O and 2 different string libraries given its C heritage, and if you make the wrong choices, performance is terrible. I'm totally sure that given some more rewriting, the C++ implementation can be made the fastest. It's just a question of how much pain you go through to find the right rewrites. All the other implementations were written in the most straightforward, simplest way possible. C++ abjectly fails the "get it done quick" criterion.]

My most pleasant surprise learning Awk was its shortcuts for reading and writing files. Like shell, there is no concept of a file handle — you don't open and close files, you just specify the filename and the VM figures it out. Even Perl, king of syntactic shorthand, doesn't have this useful feature; and even Ruby, with its elegant open()-block-cleanup construct, is clunkier. It sounds minor, but this eliminates a number of bugs you can make in scripts.

But what I most appreciate about Awk is the discourse structure of its programs: every clause is a potentially conditional action to be performed with the current record. If you want actions to be taken upon program start or exit, you declare special clauses for that. Awk manages all these features while being staying incredibly small and simple — the advantages of being a domain-specific language. I think it feels a little more like a super-flexible, index-challenged version of SQL than it does a standard scripting language. I suspect Awk's simplicity and specialization is part of why Mike Brennan was able to make Mawk so insanely fast. If your only datatypes are strings and hashes, then compile-time type inference is pretty easy.

Awk is also well-suited to the "Disk is the New Tape" era. That is, right now, hard drive sizes are rapidly growing — allowing very large datasets — but random access seek times aren't catching up. In this setting, the only way to process data is via linear scans, accessing one item of data at a time. (E.g. running variance, online learning, column stores, etc.) This is the core philosophy behind Hadoop's computation model — and Awk's. If hard drives are like tape drives, then it's worth looking in to other blast-from-the-past technologies! (Similar point about SAS, in fact.)

There are many Awk tutorials on the web. This one is decent, though I strongly recommend Ken Church's classic tutorial Unix for Poets. It shows how to do all sorts of great things with Unix text processing tools, including Awk.

All the code, results, and data can be obtained at github.com/brendano/awkspeed. I'd love to see results for more languages. And I hope someday someone tries writing an LLVM Awk — will it be even faster?

This entry was posted in Uncategorized. Bookmark the permalink.

48 Responses to *Don't MAWK AWK – the fastest and most elegant big data munging language!*

Pete Kirkham *says:*

September 10, 2009 at 7:39 am

Adding units for the timings to the blog post would help put it into context.

Adam *says:*

September 10, 2009 at 8:52 am

Cool! I'll definitely have to learn more about AWK's awesomeness.

---

brendano *says:*

September 10, 2009 at 3:29 pm

Pete – fixed, thanks. Anything else confusing?

OJ *says:*

September 10, 2009 at 4:35 pm

Two points about Java:

– It takes only a few lines to separate reading and splitting lines to different thread and have it communicate with main tread through BlockingQueue

– Java's standard string split is slow. Essentially it compiles the column separator as a regular expression and uses it every time it is called. Rolling your own split function *should* be unnecessary but having one around is useful.

For one time use these tricks are not really necessary but when you have to read several gigabytes of tab-separated data several times in a row…

The separate read/split thread could probably be made an integral part of AWK.

---

doki_pen *says:*

September 10, 2009 at 5:58 pm

You should try doing a simple ruby –help before posting such nonsense. I'm sure you missed similar features in other languages. The unix way is to pipe the file and processes stdin. No need to open and close a file. I didn't read the entire article, but if you didn't you should post all your code.

---

jc *says:*

September 10, 2009 at 6:56 pm

Err, you should read the article before calling BS. He's reading from multiple files and writing back to similarly named files. If there's a way to write to multiple files in that manner, while using some global values, I don't want to see it in shell, it would look almost as much like snoopy barf as perl.

Fletch *says:*

September 10, 2009 at 7:52 pm

Got your perl tweaked down to around 1:25s avg: http://gist.github.com/184768. Made it a bit more idiomatic modern Perl and then optimized (reduced hash lookups to once per line by stashing the value off to a lexical scalar, yadda yadda yadda).

Interesting post though; now I need to grab mawk for RHEL boxen at work and see what speed gains it gets vs the stock awk.

---

brendano *says:*
September 10, 2009 at 11:33 pm

Fletch, thanks! That's great. Obviously I don't know much perl :)

OJ, I never thought about using threads like that. Interesting. Have you implemented something like this before? And assuming you have 2 cores available, how much can it increase performance?

Todd *says:*
September 11, 2009 at 5:51 am

Actually, a Perl script can look a lot like an awk script if we exploit
some of Perl's command line arguments … see the perlrun man page
and variables … see the perlvar man page.

Building on Fletch's rewrite and writing the vocabulary to stderr
here is a two line Perl script to do the job.

```
#!/usr/bin/perl -ani_org
# USAGE: ./me file1 file2 file3 ... 2>vocab # bash style redirect of stderr

exists $jm{$F[1]} or print STDERR "$F[1]\n";
print(($im{$ARGV,$F[0]} ||= ++$i{$ARGV}).' '.($jm{$F[1]} ||= ++$j)." $F[2]\n");
```

Where '-a' causes perl to do an awk-like automatic split in to @F
'-n' causes perl to put a loop around the script
'-i_orig' cause perl to update each file "in place"
first backing up each file to file_orig
'$ARGV' is the name of the file currently being read
'$;' we are relying on the default for this not be in
any of the item names ($F[0]) or file names ($ARGV)
in the expression $im{$ARGV,$F[0]}

As a possible tweak for speed we could remove some of the hashing
by resetting %imap at the top of each file by adding one line of code.
This is a rather crude way of what Fletch did with 'my' variables.
Here we get rid of one more hash access than Fletch did.
But he could do that in his code also.

```
#!/usr/bin/perl -ani_org
# USAGE: ./me file1 file2 file3 ... 2>vocab # bash style redirect of stderr

if ($curfile ne $ARGV) { undef %imap; $i = 0; $curfile = $ARGV }
exists $jm{$F[1]} or print STDERR "$F[1]\n";
print(($im{$F[0]} ||= ++$i).' '.($jm{$F[1]} ||= ++$j)." $F[2]\n");
```

Okay, I'm stretching the definition of a line …

If one dislikes the structure of the print statements, we could
exploit some special variables
'$,' sets the output field separator
'$\' sets the output record separator
and write

```
#!/usr/bin/perl -ani_org
# USAGE: ./me file1 file2 file3 ... 2>vocab # bash style redirect of stderr
```

```
BEGIN { $, = ' '; $\ = "\n" }
if ($curfile ne $ARGV) { undef %imap; $i = 0; $curfile = $ARGV }
exists $jm{$F[1]} or print STDERR $F[1];
print(($im{$F[0]} ||= ++$i), ($jm{$F[1]} ||= ++$j), $F[2]);
```

If we rejected writing the vocabulary to stderr we add to the BEGIN statement

```
#!/usr/bin/perl -ani_org
# USAGE: ./me file1 file2 file3 ...

BEGIN {
open(VOCAB, '>vocab') or die $!;
$, = ' ';
$\ = "\n";
}
if ($curfile ne $ARGV) { undef %imap; $i = 0; $curfile = $ARGV }
exists $jm{$F[1]} or print STDERR $F[1];
print(($im{$F[0]} ||= ++$i), ($jm{$F[1]} ||= ++$j), $F[2]);
```

Finally, if we don't like the way the '-i' option works we can
add one line to do it our selves

```
#!/usr/bin/perl -an
# USAGE: ./me file1 file2 file3 ...

BEGIN {
open(VOCAB, '>vocab') or die $!;
$, = ' ';
$\ = "\n";
}
if ($curfile ne $ARGV) { # starting new file
open(STDOUT, '>', $ARGV . 'n') or die $!;
undef %imap;
$i = 0;
$curfile = $ARGV
}
exists $jm{$F[1]} or print VOCAB $F[1];
print(($im{$F[0]} ||= ++$i), ($jm{$F[1]} ||= ++$j), $F[2]);
```

Pingback: *Destillat KW37-2009 | duetsch.info - GNU/Linux, Open Source, Softwareentwicklung, Selbstmanagement, Vim ...*

David R. MacIver *says:*
September 11, 2009 at 9:49 am

As a general comment: I'm actually *really* surprised at how well Java is doing here. I've spent more time than I'd like to admit doing this sort of task in Java, and inevitably what happens when I need it to be repeatable is that I have to tear apart java.io and hand roll specific implementations, and the code typically becomes at a minimum twice as fast and usually more when I do. There's too much in memory copying and the unicode support is a real slow down if you don't need it.

On the String.split front: An easy speedup is to do static final Pattern WHITESPACE = Pattern.compile(" "); and use WHITESPACE.split(string) instead. It seemed to shave about 10 seconds off the runtime on my computer, which left it still slower than awk.

Eric Young *says:*
September 11, 2009 at 11:36 am

I've done a faster ruby version. http://gist.github.com/185234.
I'm on a linux-x86_64, 3ghz Penom II, gawk seems much faster on my box

33.8s mawk
36.3s gcc c

51.0s java
67.0s perl Fletch.pl
71.7s python
87.8s perl
95.8s nawk
101.4s gawk
114.0s gcc
133.0s ruby1.9 eay.rb
136.8s ruby1.8 eay.rb
327.6s ruby1.8
372.9s ruby1.9

OJ *says:*
September 11, 2009 at 6:41 pm

Brendano: On my dual core machine, the difference between threaded and non-threaded version is around 25%. I'm not sure why the difference is not any greater as this thing puts both CPUs near full load. I thought it could be thread communication overhead, but packing multiple lines into larger packets did not help.

It is interesting that the single thread version takes CPU load over 100%, probably due parallelized GC.

The code is available here. Uglier and more wordy than I thought.

Carlo *says:*
September 23, 2009 at 5:42 pm

I've just hit this post and I hope I'm still in time to comment on it. I've been using AWK for more than a decade now, to the point that I have written a whole application development framework for Web applications, largely based on AWK. I confirm that MAWK is lightening fast but I'm getting the impression it is no longer actively developed. I hope I'm wrong. MAWK is still affected by a few bugs that seem to have been around for years now, and nobody seems to be interested with fixing them. I have just stumbled upon this one.

Jacob *says:*
December 7, 2009 at 1:28 am

It seems to me that the issue of processing speed is almost irrelevant for data formating tasks. All tools seem fast enough. For a typical new data set, one needs to do it once and then never do it again. A lot more time is spent on writing and testing the code to accomplish this task. Based on these presumptions, the AWK implementations still beat everything else. AWK was designed for digging through formated text files and it's nice to see that it still excels at such tasks.

Haris *says:*
December 12, 2009 at 11:21 am

I am an AWK fan and I use gawk. I didn't knew that mawk it wasn't interpreted language, so I gave it a try, inspired by your article.

I did some tests but I miss certain functionality like gensub, asort, asorti functions .

Are there any mawk functions I can use as a replacement of the above ones?

---

brendano *says:*
December 12, 2009 at 5:50 pm

Not that I know of…

Keith Smith *says:*
March 4, 2010 at 10:12 am

This is kind of silly. The purpose of awk, etc is for a quick and dirty. Mawk, like dash is broken in subtle ways. Always script to standard available utilities or pay the price.

Why on earth would you use Ruby for this? That's just plain ignorant, although vanilla PHP might be interesting. Assuming your data set is formatted as you specified, C code follows. Linear search might be an issue if nelem gets overlarge, but likely not even close to a bottleneck. 140 odd lines with comments, with assumptions galore.

```
// File: 2num.c
//
// Last Modified: 2010-03-04 02:54:53
//
#include
#include
#include
#include
#include
#include
#include

#define BUFSIZE 65535
char big_buf[BUFSIZE + 1];

#define LIST_BREAK 16384
struct list_struct {
    char **list;
    int size,ix,new;
} list1,list2;

int in_fd,out_fd,vocab;

//-----------------------------------------------------------------------
// CHECK_LIST
// Simple Linear search is faster up to around 16K elements
//-----------------------------------------------------------------------
check_list(struct list_struct *lp, char *val) {
    char **p;
    int ix;
    for(p = lp->list,ix = 0; ix ix && lp->list[ix] != NULL; ix++) {
        if(strcmp(lp->list[ix],val) == 0) {
            return(ix + 1);
            break;
        }
    }
    // Not there
    if(lp->ix > lp->size) {
        lp->list = realloc(lp->list,lp->size + LIST_BREAK);
    }
    lp->list[lp->ix++] = strdup(val);
    lp->new = 1;
    return(lp->ix);
}

//-----------------------------------------------------------------------
// PROCESS
//-----------------------------------------------------------------------
void process(char **vals) {
    int n1,n2;
    char obuf[32];

    n1 = check_list(&list1,vals[0]);
    n2 = check_list(&list2,vals[1]);
    if(list2.new) { // Optmizeme into a buffer
        write(vocab,vals[1],strlen(vals[1]));
        write(vocab,"\n",1);
        list2.new = 0;
    }
    // Optimize into a buffer ...
    sprintf(obuf,"%d %d %s\n",n1,n2,vals[2]);
    write(out_fd,obuf,strlen(obuf));
}

//-----------------------------------------------------------------------
// PARSE
//-----------------------------------------------------------------------
char **parse(char *line) {
    static char *p[3];
    p[0] = line;
    p[1] = index(line,' ');
    if(p[1] == NULL) {
        p[1] = "";
        p[2] = "";
        return(p);
    }
    *(p[1])++ = '';
    p[2] = index(p[1],' ');
    if(p[2] == NULL) {
        p[2] = "";
    } else {
        *(p[2])++ = '';
    }
    return(p);
}

//-----------------------------------------------------------------------
// MAIN
//-----------------------------------------------------------------------
main(int argc, char *argv[]) {
    int ix, iy, bytes_read,buflen;
    char *p,*q, **vals;
    char buf[4096];

    list1.list = malloc(LIST_BREAK);
    list1.size = LIST_BREAK;
    list1.ix = 0;
    list1.new = 0;

    list2.list = malloc(LIST_BREAK);
    list2.size = LIST_BREAK;
    list2.ix = 0;
    list2.new = 0;

    vocab = open("vocab",O_WRONLY|O_CREAT|O_TRUNC,0666);
    if(vocab == -1) {
        perror("Open vocab");
        exit(-1);
    }
    for(ix = 1; ix  0;) {
            // Fill the buffer
            for(buflen = 0; buflen < BUFSIZE;) {
                bytes_read = read(in_fd, big_buf + buflen,BUFSIZE - buflen);
                if(bytes_read  0 && p != NULL;) {  // Parse off lines and process them
                q = index(p,'\n');
                if(q == NULL) { // Read Some More
                    buflen = strlen(p);
                    memcpy(big_buf,p,buflen);
                    break;
                }
                *q++ = '';
                vals = parse(p);
                process(vals);
                p = q;
            }
        }
        close(in_fd);
        close(out_fd);
```

```
            list1.ix = 0;
        }
        close(vocab);
    }
```

Pingback: *MAWK does not support length(array)? - The UNIX and Linux Forums*

Karl *says:*

September 24, 2010 at 8:30 pm

I have been using mawk for nearly a year now for very large data sets. I didn’t believe it was nearly 8 times faster than nawk, until I tested it myself. I’ve used nawk and C for 13 and 18 years. I had switched to C to process my particularly large data sets, because nawk was too slow. I switched to mawk because I have found it a little faster than C, and much faster to code and debug.

Nawk has some extra functions such as array sort, but fortunately I haven’t needed them for my big data. The speed increase of mawk over nawk is incredible.

I am super pleased with mawk. Long live mawk!

Anders *says:*

October 27, 2010 at 8:13 am

I compared mawk to gawk and mawk is 4 times faster than gawk. I tested on a file with about one million lines. (6 seconds versus 24 seconds).

length(array) is easy to do yourself:

for (key in array){
len++;
}

and it is obviously fast since I do it in the above mawk program.
Very well done Mike Brennan!

Derek Schrock *says:*

November 2, 2010 at 1:50 am

You should try COBOL.

frank gleason *says:*

November 15, 2010 at 10:20 pm

Do you have a link to the large datasets that causes mawk to give incorrect output? I would like to see if mawk can be fixed.

Nobody *says:*

July 6, 2011 at 12:40 pm

2num.cc uses std::endl for end of lines, but this also flushes buffer.
<< '\n' may be much faster.

Nobody *says:*

July 6, 2011 at 2:57 pm

gawk will be faster, if you redirect its output to /dev/null, try

gawk -f 2num.awk *.p > /dev/null

Bill Gates *says:*
October 5, 2011 at 1:14 am

@Nobody: Try a long record (read: no newlines) and use gsub() on that record …

schoolyeolde *says:*
October 6, 2011 at 8:07 pm

you forgot to factor in the milliseconds it takes to type the LOC.
e.g. 3 vs. 10+ lines

schoolyeolde *says:*
October 6, 2011 at 8:33 pm

or the time installing, configuring and maintaining external libraries (e.g. perl, python, etc.). or the download and compile time for the code (size), if the programs are not part of the base system. the time to load the binary into memory if it's not already cached (again, size). etc.

with these experiments, we assume that all this preliminary stuff is taken care of. that may not be true for every user in every case.

mss *says:*
November 12, 2011 at 6:38 am

realizing I'm late to the game on this thread …

sh most certainly has file handles, and it is recommended to use them to avoid opening a file-handle repeadedly to any particular file (for logging, for example):

```
# open on fd7 for writing
exec 7> /my/logfile
for i in 1 2 3;
do
   # output to open fd
    echo "$i" >&7
done
#close
exec 7> /my/logfile
done

and allows for other usage of stdout in the for loop ...
exec 7> /my/logfile
{
    for i in 1 2 3;
    do
        echo "$i" >&7
        echo "something i don't want to go to the log"
    done
} | ...
exec 7<&-
```

Leonid Volnitsky *says:*
January 11, 2012 at 8:47 am

There is something called SCC . It is several times faster than MAWK in my benchmarks. But it is alpha version and needs GCC-4.7 for runtime.

Brendan O'Connor *says:*

January 11, 2012 at 4:23 pm

Leonid, nice work, this looks cool! Thanks for posting.

---

neil *says:*

August 9, 2012 at 1:40 am

little late here, but C++ using STL and templates is not “C-ish”. Would like to see a benchmark with an actual C version.

Tomer Altman *says:*

November 9, 2012 at 8:28 pm

I’ve been able to shave ~30% off of the script’s execution time relative to gawk, by compiling the following version of your script using awka (awka.sf.net/index.html). The changes in the script were done based on advice on the Awka site on how to modify scripts for better performance. But, amazingly, MAWK, is still faster than the compiled version using AWK (using gcc -O3). YMMV.

—

```
## Use this pattern/action to get rid of the I associative array:
FNR == 0 { i_count = 0 }

## We note that the array index access occurs for every line in the file,
#so we consolidate the following pattern/action pairs below as an
#optimization for the awka compiler:

# !imap[FILENAME, $1] {
# imap[FILENAME, $1] = ++i_count
# }

# !jmap[$2] {
# vocab_word = $2
# jmap[vocab_word] = ++J
# }

## Execute for each line:
{ ## First, assign positional parameters to save on multiple array lookups:
item=$1
feature=$2
## Secondly, assign imap and jmap array lookups to scalars:
current_imap = imap[FILENAME,item]
current_jmap = jmap[feature]
## Finally, print updated value if needed, otherwise, print current
## value of *maps:
print ! current_imap ? imap[FILENAME,item]=++i_count : current_imap,
! current_jmap ? jmap[feature]=++J : current_jmap,
$3 > (FILENAME “n”) }

## We save printing of the smaller “vocab” file to the end:
END{ for ( i in jmap ) print i > “vocab” }
```

Pingback: *Don’t MAWK AWK – the fastest and most elegant big data munging language (?) | My Daily Feeds*

Pingback: *Mawking AWK with Lisp | Irreal*

---

jcs *says:*

November 29, 2012 at 4:07 pm

Hi Brendan,

I ran the benchmark with Lisp and several of the other languages that you used. Things shuffled about a bit but `mawk` is still the clear winner. You can see my results at my blog

---

steck *says:*

May 3, 2013 at 6:09 pm

I love using AWK for data munging tasks.

Unfortunately, mawk is limited to 32767 fields per line, and I'm working with datasets that have 12x that number of fields on some lines.

Gert *says:*

December 26, 2013 at 10:07 pm

You need to recompile mawk (see next post).

Gert *says:*

December 26, 2013 at 9:40 pm

@ steck:

Get mawk source code from: http://invisible-island.net/mawk/

By default, mawk with the following settings will be compiled:

```
$ ./mawk -W version
mawk 1.3.4 20130803
Copyright 2013, Thomas E. Dickey
Copyright 1996, Michael D. Brennan

internal regex
compiled limits:

max NF 32767
sprintf buffer 1020
```

To change the number of fields mawk can handle, you need to change sizes.h, so the compiler will use FBANK_SZ = 16384 and FB_SHIFT = 14:

```
$ diff -U 10 sizes.h.orignal sizes.h
--- sizes.h.orignal 2013-12-26 22:18:00.008386400 +0100
+++ sizes.h 2013-12-26 22:36:58.300493000 +0100
@@ -88,22 +88,22 @@
#define EVAL_STACK_SIZE 256 /* initial size , can grow */

/*
* FBANK_SZ, the number of fields at startup, must be a power of 2.
* Also, FBANK_SZ-1 must be divisible by 3, since it is used for MAX_SPLIT.
*
* That means that FBANK_SZ can be 256, 1024, 4096, 16384, etc. (growing by
* a factor of four for each next possible value).
*/
#if 1
-#define FBANK_SZ 256
-#define FB_SHIFT 8 /* lg(FBANK_SZ) */
+#define FBANK_SZ 16384
+#define FB_SHIFT 14 /* lg(FBANK_SZ) */
#else
#define FBANK_SZ 1024
#define FB_SHIFT 10 /* lg(FBANK_SZ) */
#endif
#define NUM_FBANK 128 /* see MAX_FIELD below */
```

```
#define MAX_SPLIT (FBANK_SZ-1) /* needs to be divisble by 3 */
#define MAX_FIELD (NUM_FBANK*FBANK_SZ - 1)
/*
* mawk stores a union of MAX_SPLIT pointers and MIN_SPRINTF characters.
```

Configure and build mawk:

```
./configure
make clean
make
```

Now mawk is able to handle 2097151 fields per line:

```
$ ./mawk -W version
mawk 1.3.4 20130803
Copyright 2013, Thomas E. Dickey
Copyright 1996, Michael D. Brennan

internal regex
compiled limits:
max NF 2097151
sprintf buffer 65532
```

Andrew *says:*

January 10, 2014 at 7:01 pm

I've loved reading this comment set. It's old (in internet years) but very informative. Thanks for all the commentary. I hope there might be a solution to my problem. A lot of my work requires summing over long periods of data, such as the total GET'd data on a busy web cluster in a month. Mawk is fantastic– I'm getting about 3.5x the performance I saw in gawk.
The problem I'm seeing is the 2147483647 max on %d-formatted numbers. Here's output from the same script, where the first line is **sprintf %d**'d and the second is simply **print**'d:

```
total_size, total_count, average: 2147483647 50586 2493242
total_size, total_count, average: 1.26123e+11 50586 2.49324e+06
```

Sadly, the exponential notation doesn't work for the report's audience. Is there some workaround for this? With careful direction I don't mind editing the source code, but it's really not my forte.

Again, thanks for all the useful comments.

Gert *says:*

March 16, 2014 at 10:41 pm

The developer of the upstream mawk says it is a limitation of the printf funtion (to keep it fast):

> This is a known limitation: mawk's format for %d is limited by the format.
> The limitation is done to improve performance.
>
> You can get more precision using one of the floating formats (and can construct
> one which prints like a %d, e.g., by putting a ".0″ on the end of the format).
http://code.google.com/p/original-mawk/issues/detail?id=23

---

brendano *says:*

January 11, 2014 at 7:24 pm

Unfortunately, limitations like this seem to keep cropping up in mawk…

Ronald Loui *says:*

May 11, 2014 at 9:06 pm

```
FILENAME != lastf { lastf = FILENAME; delete imap; I=0 }
!imap[$1] { imap[$1] = ++I }
!jmap[$2] { jmap[$2] = ++J }
{ print imap[$1], jmap[$2], $3 > (lastf "n") }
END { for (v in jmap) print v > "vocab" }
```

will save you 10-30% in my gawk, depending on the length of your files.

Now, if Arnold Robbins would give us back the pre-allocate hash size option at the command line, we could get that faster using extra wide hashes.

Note that I am taking advantage of the serial file processing. I am a huge fan of awk/gawk, data pipieline processing, and pragmatics, such as how much time it takes you to write, debug, maintain, explain, etc.

Since Stallman showed me gawk in 92, I've been a huge fan. Good to see awk/nawk/gawk/mawk get some well deserved respect. Solaris nearly killed nawk by giving us the worst implementation in history.

A few language extensions could speed this up even more. Array-in and array-out would reduce the file-i/o overhead. ENDFILE and BEGINFILE would remove an annoying conditional. For long lines, a parse-to-n-and-quit, like the split in bash and perl, would also save time, since $n, n>3, is never referenced. It's possible that a two-stage "i in array" test would be faster than hash-collision-chain traversal, where the first stage is a bloom filter.

In fact, this is another 10% faster…

```
BEGIN {
for (ifile=2; ifile in ARGV; ifile++) {
inf = ARGV[ifile]
outf = inf "n"
delete imap
I=0
while (getline outf
}
}
for (v in jmap) print v > "vocab"
}
```

awk "experts" abhor the use of BEGIN for the main loop, but the fact is, you get a lot more control, you can process multiple streams, you can increase readability and correctness, AND you can popularize the language for more general scripting use.

So happy to see discussions like this.

Ronald Loui *says:*

May 11, 2014 at 9:08 pm

Sorry, but it seems my getline code and indenting has been eaten by HTML. Trying again:

```
BEGIN {
for (i=2; i in ARGV; i++) {
inf = ARGV[i]
outf = inf "n"
delete imap
I=0
while (getline < inf) {
if (!imap[$1]) imap[$1] = ++I
if (!jmap[$2]) jmap[$2] = ++J
print imap[$1] " " jmap[$2] " " $3 > outf
}
}
for (v in jmap) print v > "vocab"
}
```

Ronald Loui *says:*

May 11, 2014 at 9:24 pm

Well, you get the idea…

Oddly, my mawk is not faster with these rewrites:

```
loui@loui-desktop:~/gawk$ mawk -W version
mawk 1.3.3 Nov 1996, Copyright (C) Michael D. Brennan

compiled limits:
max NF 32767
sprintf buffer 1020
loui@loui-desktop:~/gawk$ awk -W version
GNU Awk 3.1.8
Copyright (C) 1989, 1991-2010 Free Software Foundation.
```

Ronald Loui *says:*
May 11, 2014 at 9:30 pm

Come to think of it, you can do:

for (ii in imap) delete imap[ii]

instead of delete imap,

which should keep the largest allocation of imap's hash, and indeed gives you a bit more gawk speed.

Eric *says:*
July 22, 2014 at 4:56 pm

I stopped at reading c++ is 1.3x times faster than awk while java is 6.4x times. Do you seriously suggest that c++ is 5x times SLOWER than java?

Pingback: *AWK for Human Beings | Thoughts and Scribbles | MicroDevSys.com*
