Data Science in Ten Minutes (Homework solutions)

Spencer Tipping

May 24, 2018

Contents

1 Linux

1.1 `cat` as an ELF file

`read()` returns the number of bytes read into `%rax`, returning `0` on EOF and `-errno` on error. (TODO: verify the details)

```
TODO
```

2 Wikipedia

2.1 Tenth most common word

First with `ni` using 24x pipeline scaling for the word extraction loop:

```
$ ni enwiki* rp'/<text/../<\/text>/' S24F'/\{\{[^}]*\}\}|<[^>]+>|&\w+;/'FWpF_ \
     riA[/usr/share/dict/words] gc0 r-9 r1
```

The same algorithm implemented with standard UNIX tools (but without scaling):

```
$ pv enwiki* \
  | pbzip2 -dc \
  | awk '/<text/,/<\/text>/ {print}' \
  | sed -r 's/\{\{[^}]*\}\}|<[^>]+>|&\w+;//g; s/\W+/\n/g' \
  | perl -ne 'BEGIN { open my $fh, "< /usr/share/dict/words";
                      ++$words{$_} while <$fh> }
              print if exists $words{$_}' \
  | sort \
  | uniq -c \
  | sort -rn \
  | tail -n+10 \
  | head -n1
```

2.2 Disk-backed sort

TODO

2.3 Measuring overhead of integer parsing

Let's mock up a datasource here. The specific integer value won't matter very much; mainly we need to get the shape of the data right. If we generate numbers between 1 and 11 randomly, we should get roughly the right distribution of number lengths.

I need to bulk-generate some data and repeatedly rewrite it; this way we know the data source isn't itself the bottleneck. I'm doing this by asking ni to store a string containing 10,000 TSV rows, which should come out to between 100KB and 1MB.

```
$ tsv() {
    ni ::chunk[nE4p'r int rand(11), int rand(11)' \
               w[ enwiki* r/\<title/F/[\<\>]/fC ] ] \
       npchunk | pv
  }

$ tsv > /dev/null                       # verify source speed: ~800MB/s
$ tsv | perl -ne '$x += length'         # test line-split only: ~70MB/s
$ tsv | perl -ne 'my ($web, $other, $title) = split /\t/;
                  $x += length'         # line + TSV: ~22MB/s
$ tsv | perl -ne 'my ($web, $other, $title) = split /\t/;
                  $x += $web + $other'  # line + TSV + int parse: ~19MB/s
```

We can use units to get a percentage.

```
$ units -t '1/(19MB/s) / (1/(22MB/s))' percent
115.78947           # integer parsing adds ~16% overhead
```

2.4 pv vs cat vs dd

pv is faster than cat and dd (even across different block sizes; I couldn't find one that got close to pv speed):

```
$ dd if=/dev/zero bs=1048576 count=8192 | pv > /dev/null
8589934592 bytes (8.6 GB, 8.0 GiB) copied, 3.90754 s, 2.2 GB/s

$ dd if=/dev/zero bs=1048576 count=8192 | cat > /dev/null
8589934592 bytes (8.6 GB, 8.0 GiB) copied, 5.23577 s, 1.6 GB/s

$ dd if=/dev/zero bs=1048576 count=8192 | dd bs=1048576 > /dev/null
8589934592 bytes (8.6 GB, 8.0 GiB) copied, 5.23577 s, 1.6 GB/s
```

As for why this is, let's take a look at the source:

- cat copy loop
- dd copy loop
- pv copy loop

Both dd and cat have the same structure: read a buffer of data, then write it back out. This seems optimal, but it isn't: if all we're doing is shuttling stuff from stdin to stdout, why copy anything into virtual memory at all? The Linux kernel provides the splice syscall to bypass the extra copy, and that's exactly how pv gets its advantage.

2.5 Split Wikipedia

split has an option that lets you specify a compressor, which is useful here. The general idea would be something like this:

```
$ pbzip2 -dc enwiki* | split [options] --filter='gzip > $FILE'
```

split gives us the -l option to indicate how many lines we want per split output, but -C is more convenient here (write whole lines until a specified number of bytes). It's important to preserve line breaks in general so we avoid splitting the citations themselves.

As far as performance goes, it's going to depend on a few factors:

- Disk seek time
- Disk throughput
- Number of CPUs
- Which version of libc and core utilities you're using[1]

[1] You can see the difference if you try the alpine docker image; it uses a different libc that lacks most of the optimizations in glibc.

The general idea is that we want to balance CPU and disk load.

If we have 24 processors, it's tempting to conclude that we would want no more than 24 files; but that's not quite true. We probably want substantially more than 24 files because not all of them will run equally fast and we want to keep all of the CPU cores busy. The only real trade here is seek/process-start overhead vs CPU downtime. If we estimate that total at 10ms/file, then anything with more than about 240ms of processing time per file is more efficient than an average CPU utilization of 23/24.

The OS will do a decent job of scheduling disk reads efficiently, but we still need to consider the amount of fragmentation we're introducing and how well the disk(s) can keep the CPUs saturated. This depends a lot on the hardware. If you're on a laptop with one mobile HDD, you're likely to want a high compression ratio because you'll quickly become IO-bound. If you have a single SSD, you would want the opposite: seeks are nearly free and IO throughput will easily saturate several processors for most workflows. If you're on a server with a lot of CPUs and RAID-0, 5, or 6 across a dozen disks, you might want something middle of the line like `gzip`.

LZ4 will make some decent progress on Wikipedia text simply due to the redundancy in English text; it's worth using it rather than keeping the files uncompressed. The real question is whether we need Huffman or LZMA to get more mileage. Before we get into that, let's talk about the downstream process.

TODO