Data Science in Ten Minutes[*]

Spencer Tipping

May 24, 2018

Contents

Introduction

Data science is not machine learning. There is no machine learning in this guide, because machine learning is the wrong answer to most problems you're likely to run into. I've never deployed a machine learning system, although I've debugged several misbehaving ones (and will cover how to do that).

A lot of the code examples assume you're running Linux because Linux is the go-to platform for medium/big data processing. It's also ideal for small data due to optimizations in core utilities like `sort`.[1] If you're running a non-Linux OS and want to try stuff, you have a few options:

[*]for extremely large values of ten

[1]GNU `sort` will compress temporary files, unlike the `sort` that ships with OSX.

- Install Docker and create a container from the `ubuntu:18.04` image or similar (I'll assume you have this setup)
- Run a VM like VirtualBox and install Ubuntu Desktop inside it
- Rent a free tier Amazon EC2 instance (you can use the free-tier `t2.nano` or `t2.micro` for data science, and they're ideal for learning because they're resource-constrained)
- Buy a cheap rack server from eBay and drop Ubuntu Server on it

Realistically, you probably won't want to use OSX or Windows for distributed programming: your binaries won't be portable to cluster machines, and you'll be fighting with things like case-insensitive filesystems, slower core utilities, and general inconsistencies that will increase debugging time.[2]

Examples

This guide comes with example code in the original repository. Where necessary, I've included package installation commands required to install dependencies on Ubuntu 18.04. Any hard dependencies are pretty boring and easy to install; you won't need anything like the RVM or any custom libraries.

[2]Some of this is a non-issue for JVM processes, e.g. Hadoop and Spark, but a lot of data science is best done with native tools that are more powerful, use less memory, and have much lower iteration time.

1 Linux

Oh yes, we are totally going here. Here's why.

Backend programs and processing pipelines and stuff (basically, "big data" things) operate entirely by talking to the kernel, which, in big-data world, is usually Linux; and this is true regardless of the language, libraries, and framework(s) you're using. You can always throw more hardware at a problem[3], but if you understand system-level programming you'll often have a better/cheaper option.

Some quick background reading if you need it for the homework/curiosity:

- How to write a UNIX shell
- How to write a JIT compiler
- ELF executable binary spec (more readable version on Wikipedia, and this StackOverflow answer may be useful)
- Intel machine code documentation

1.1 Virtual memory

This is one of the two things that comes up a lot in data science infrastructure. Basically, any memory address you can see is virtualized and may not be resident in the RAM chips in your machine. The kernel talks to the MMU hardware to maintain the mapping between software and hardware pages, and when the virtual page set overflows physical memory the kernel swaps them to disk, usually with an LRU strategy.

Here's where things get interesting. Linux (and any other server OS) gives you a `mmap` system call to request that the kernel map pages into your program's address space. `mmap`, however, has some interesting options:

- `MAP_SHARED`: map the region into multiple programs' address spaces (this reuses the same physical page across processes)
- `MAP_FILE`: map the region using data from a file; then the kernel will load file data when a page fault occurs

`MAP_FILE` is sort of like saying "swap this region to a specific file, rather than the shared swapfile you'd normally use." The implication is important, though: *all memory mappings go through the same page allocation cache,* and any page fault has the potential to block your program on disk IO. Clever data structures like Bloom filters, Count-min sketches, and so forth are all designed to give you a way to trade various degrees of accuracy for a much smaller memory footprint.

Sometimes you won't have any good options within the confines of physical RAM, so you'll end up using IO devices to supply data; then the challenge

[3]Until you can't

becomes optimizing for those IO devices (SSDs are different from HDDs, for instance). I'll get to some specifics later on when we talk about sorting, joins, and compression.

1.2 File descriptors

This is the other thing you need to know about.

Programs don't typically use `mmap` for general-purpose IO. It's more idiomatic, and sometimes faster, to use `read` and `write` on a file descriptor. Internally, these functions ask the kernel to copy memory from an underyling file/socket/pipe/etc into mapped pages in the address space. The advantage is cache locality: you can `read` a small amount of stuff into a buffer, process the buffer, and then reuse that buffer for the next `read`. Cache locality does matter; for example:

```
# small block size: great cache locality, too much system calling overhead
$ dd if=/dev/zero count=262144 bs=32768 of=/dev/null
262144+0 records in
262144+0 records out
8589934592 bytes (8.6 GB, 8.0 GiB) copied, 0.774034 s, 11.1 GB/s

# medium block size: great cache locality, insignificant syscall overhead
$ dd if=/dev/zero count=8192 bs=1048576 of=/dev/null
8192+0 records in
8192+0 records out
8589934592 bytes (8.6 GB, 8.0 GiB) copied, 0.574597 s, 14.9 GB/s

# large block size: cache overflow, insignificant syscall overhead
$ dd if=/dev/zero count=2048 bs=$((1048576 * 4)) of=/dev/null
2048+0 records in
2048+0 records out
8589934592 bytes (8.6 GB, 8.0 GiB) copied, 1.14022 s, 7.5 GB/s
```

This makes sense considering the processor hardware:

```
$ grep cache /proc/cpuinfo
cache size : 3072 KB
cache_alignment : 64
cache size : 3072 KB
cache_alignment : 64
cache size : 3072 KB
cache_alignment : 64
cache size : 3072 KB
cache_alignment : 64
```

1.3 Concurrency and FIFOs

When you say something like `cat file | wc -l`, `cat`'s `stdout` (file descriptor 1) maps to the write-end of a kernel FIFO pipe and `wc`'s `stdin` (fd 0) maps to the read end of that same FIFO. `wc`'s `stdout` is the same as your shell's `stdout`: it points to the terminal device.

This raises an interesting question: what happens if two separate processes write to the same FIFO device? Those processes could be running on separate processors, which means a race condition could theoretically arise. Roughly speaking, the kernel applies a couple of rules to the situation:

1. Each device has a well-defined timeline, so `write` calls are serialized per device. I'm not sure how the kernel breaks ties, but it probably doesn't matter very much.

2. Writes of `PIPE_BUF`[4] or fewer bytes are atomic; that is, you're guaranteed that those bytes will all be grouped together in the output.

3. Once you've written data, it's committed; there's no buffering or undoing a `write` at the system call level.[5]

1.4 Pulling this together: let's write a program

...in machine language. For simplicity, let's write one that prints `hello world` and then exits successfully (with code 0).

This is also a good opportunity to talk about how we might generate and work with binary data with things like fixed offsets. Two simple functions for this are `pack()` and `unpack()`, variants of which ship with both Perl and Ruby.

The first part of any Linux executable is the ELF header, usually followed directly by a program header; here's what those look like as C structs for 64-bit executables (reformatted slightly, and with docs for readability):

```
typedef struct {
  unsigned char e_ident[16];    // 0x7f, 'E', 'L', 'F', ...
  uint16_t      e_type;         // ET_EXEC = 2 for executable files
  uint16_t      e_machine;      // EM_X86_64 = 62 for AMD64 architecture
  uint32_t      e_version;      // EV_CURRENT = 1
  uint64_t      e_entry;        // virtual address of first instruction
  uint64_t      e_phoff;        // file offset of first program header
  uint64_t      e_shoff;        // file offset of first section header
  uint32_t      e_flags;        // always zero
  uint16_t      e_ehsize;       // size of the ELF header struct (this one)
  uint16_t      e_phentsize;    // size of a program header struct
```

[4] 4096 on my system, but it can be as low as 512. You can find this value using `getconf -a | grep PIPE_BUF`.

[5] Devices sometimes do their own buffering, e.g. for network connections, but you can't access these buffers.

```
  uint16_t      e_phnum;        // number of program header structs
  uint16_t      e_shentsize;    // size of a section header struct
  uint16_t      e_shnum;        // number of section header structs
  uint16_t      e_shstrndx;     // string table linkage
} Elf64_Ehdr;

typedef struct {
  uint32_t p_type;              // the purpose of the mapping
  uint32_t p_flags;             // permissions for the mapped pages (rwx)
  uint64_t p_offset;            // file offset of the first byte of data
  uint64_t p_vaddr;             // virtual memory offset of the data (NB below)
  uint64_t p_paddr;             // physical memory offset (usually zero)
  uint64_t p_filesz;            // number of bytes from the file
  uint64_t p_memsz;             // number of bytes to be mapped into memory
  uint64_t p_align;             // segment alignment
} Elf64_Phdr;
```

If we want a very minimal executable, here's how we might write these headers from Perl:

```
# elf-header.pl: emit an ELF binary and program header to stdout
use strict;
use warnings;

print pack('C16 SSL',
           0x7f, ord 'E', ord 'L', ord 'F',
           2, 1, 1, 0,
           0, 0, 0, 0,
           0, 0, 0, 0,

           2,                           # e_type    = ET_EXEC
           62,                          # e_machine = EM_X86_64
           1)                           # e_version = EV_CURRENT

    . pack('QQQ',
           0x400078,                    # e_entry = 0x400078
           64,                          # e_phoff
           0)                           # e_shoff

    . pack('LSS SSSS',
           0,                           # e_flags
           64,                          # e_ehsize
           56,                          # e_phentsize
           1,                           # e_phnum

           0,                           # e_shentsize
```

```
          0,                            # e_shnum
          0)                            # e_shstrndx

   . pack('LLQQQQQQ',
          1,                            # p_type = PT_LOAD (map a region)
          7,                            # p_flags = R|W|X
          0,                            # p_offset (must be page-aligned)
          0x400000,                     # p_vaddr
          0,                            # p_paddr
          0x1000,                       # p_filesz: 4KB
          0x1000,                       # p_memsz: 4KB
          0x1000);                      # p_align: 4KB
```

Now we can generate the ELF header:

```
$ sudo apt install perl                 # if perl is missing
$ perl elf-header.pl > elf-header
```

You can verify that the header is correct using `file`, which reads magic numbers and tells you about the format of things:

```
$ sudo apt install file
$ file elf-header
elf-header: ELF 64-bit LSB executable, x86-64, version 1 (SYSV), statically
linked, corrupted section header size
```

Awesome, now let's get into the machine code.

1.5 `hello world` in machine code

The first thing to note is that we're talking directly to the kernel here. Our ELF header above is very minimalistic, with no linker instructions or anything else to complicate things. So we have none of the usual libc functions like `printf` or `exit`; it's up to us to define those in terms of Linux system calls.

There are two calls we'll use for this: `write(2)` and `exit(2)`. You can view the documentation for these using `man`:

```
$ sudo apt install man manpages-dev
$ man 2 write
$ man 2 exit
```

We'll need to pass arguments in registers to match the kernel calling convention. I'll spare you the gory details and cut straight to the machine code:

```
# elf-hello.pl: emit machine code for hello world, then exit successfully
use strict;
use warnings;
```

```
print pack('H*', join '',
  '4831c0',                           # xorq %rax, %rax
  'b001',                             # movb $01, %al (1 = write syscall)
  'e80c000000',                       # call %rip+12 (jump over the message)
  unpack('H*', "hello world\n"),      # the message
  '5e',                               # pop message into %rsi (buf arg)
  'ba0c000000',                       # movl $12, %rdx (len arg)
  '48c7c701000000',                   # movl $1, %rdi (fd arg)
  '0f05',                             # syscall instruction

  '4831c0',                           # xorq %rax, %rax
  'b03c',                             # movb $3c, %rax (3c = exit syscall)
  '4831ff',                           # xorq %rdi, %rdi (exit code arg)
  '0f05');                            # syscall instruction
```

Now we can build the full executable, verify it, and run:

```
$ sudo apt install nasm
$ perl elf-hello.pl | cat elf-header - > elf-hello
$ chmod 755 elf-hello
$ tail -c+121 elf-hello | ndisasm -b 64 -
00000000  4831C0            xor rax,rax
00000003  B001              mov al,0x1
00000005  E80C000000        call 0x16
0000000A  68656C6C6F        push qword 0x6f6c6c65   # corruption from message
0000000F  20776F            and [rdi+0x6f],dh       # (which we jumped over)
00000012  726C              jc 0x80
00000014  640A5EBA          or bl,[fs:rsi-0x46]
00000018  0C00              or al,0x0
0000001A  0000              add [rax],al
0000001C  48C7C701000000    mov rdi,0x1             # now we're back on track
00000023  0F05              syscall
00000025  4831C0            xor rax,rax
00000028  B03C              mov al,0x3c
0000002A  4831FF            xor rdi,rdi
0000002D  0F05              syscall
```

The moment we've been waiting for:

```
$ ./elf-hello
hello world
$ echo $?                             # check exit status
0
```

Whether through static/dynamically-linked libraries, JIT, or anything else, this is the exact mechanism being used by any program that performs IO of any sort: the program lives in a completely virtual world and interacts with the kernel using the `0f05` syscall instruction, referring to virtual addresses in the process. Efficient data science is ultimately about maximizing the throughput you're getting from the system calls you make (which, in practice, means choosing languages and libraries that make this easy to do for your application).

1.6 Homework, if that's your thing

1. Write an ELF Linux executable that consumes data from `stdin` and writes that data to `stdout`, then exits successfully. In other words, `cat` without file support.

2. Use `pack()` to produce a RIFF WAV file containing a 440Hz sine wave for ten seconds. It may be helpful to use `unpack()` to inspect the headers of existing WAV files because it's challenging to find detailed documentation of the format.

3. Problem (2), but have no more than 256 bytes of string data resident at any given moment.

2 Wikipedia

You can download the full English language Wikipedia as a giant bzip2-compressed XML file. For reference, here's the exact torrent file I downloaded. It's about 14GB compressed, which is not big data by any means; you could process this on a Raspberry Pi with a 64GB SD card if you wanted to.[6]

While that's downloading, let's take a moment to talk about compression formats.

2.1 Compression

There are a few standard, general-purpose data compressors you're likely to encounter regularly:[7]

TODO: use wikipedia data for the table below, obviously

Compressor	Compression speed	Decompression speed	Efficiency
`xz`	4MB/s	200MB/s TODO	High
`bzip2`	8MB/s	**24MB/s/core**	High-ish
`gzip`	23MB/s	120MB/s	Medium
`lzo`	200MB/s	300MB/s TODO	Low
`lz4`	240MB/s	800MB/s TODO	Low

If you take one thing away from this table, it's *don't use bzip2*. `bzip2` is horrible, even though the algorithm is pretty cool. If you are ever cursed with a bzip2 file, you can accelerate decompression by parallelizing it across multiple cores using `pbzip2`, which is installable under Ubuntu using `sudo apt install pbzip2`.[8]

Roughly speaking, here's how these compressors operate:[9]

Compressor	Structure
`xz`	Large-dictionary LZ77 + LZMA + statistical prediction
`bzip2`	RLE + BWT + MTF + RLE + Huffman + bit-sparse
`gzip`	LZ77 + Huffman
`lzo`	Dictionary
`lz4`	Dictionary

It's worth knowing the broad strokes because the nature of the data will impact compression performance, both in space and time. For example, the

[6]Be careful with SD cards and Flash storage in general; if you write the memory too many times you'll destroy the drive. I'll mention this hazard anytime I have an IO-intensive process.

[7]I ran these tests by compressing an infinite stream of copies of the ni repository, which is large enough to overflow any buffers used by these algorithms. The exact script template was `ni ::self[//ni] npself zx9 zn`.

[8]`bzip2` is atypical for supporting parallel decompression. Most are designed to be strictly serial because this design tends to produce better compression ratios.

[9]And here's some background on the theory, if that's of interest

Reddit comments dataset contains a bunch of identical-schema JSON objects that look roughly like this (reformatted for readability):

```
{ "author":"CreativeTechGuyGames",
  "author_flair_css_class":null,
  "author_flair_text":null,
  "body":"You are looking to create a reddit bot? You will want to check out
          the [reddit API](https://www.reddit.com/dev/api). Many people do
          this in Python and there are many tutorials on the internet showing
          how to do so.",
  "can_gild":true,
  "controversiality":0,
  "created_utc":1506816001,
  "distinguished":null,
  "edited":false,
  "gilded":0,
  "id":"dnqik29",
  "is_submitter":false,
  "link_id":"t3_73if9c",
  "parent_id":"t1_dnqihqz",
  "permalink":"/r/learnprogramming/comments/73if9c/how_i_would_i_go_around_making_something_
  "retrieved_on":1509189607,
  "score":1,
  "stickied":false,
  "subreddit":"learnprogramming",
  "subreddit_id":"t5_2r7yd" }
```

A nontrivial amount of the bulk in this file is stored in the JSON field names, so dictionary encoding alone is likely to save us a nontrivial amount of space. LZ4 is worthwhile here just for that, and would almost certainly be faster than reading directly from the underlying IO device.

Sometimes you need Huffman encoding, though; for example, random ASCII floats don't have enough repetition to behave well with dictionary compressors. On my test case `gzip` gets about 4x better compression than `lz4`, and that might justify preferring it to LZ4 over slow IO devices even if it creates a CPU bottleneck.

I generally start with `gzip` at its default level and change algorithms later if I need to.

2.2 OK, back to Wikipedia

So we have 14GB of bzip2 data:

```
$ ls -lh enwiki-20170820-pages-articles.xml.bz2
-rw-rw-r-- 1 114 122 14G May 20 00:38 enwiki-20170820-pages-articles.xml.bz2
```

What do we do with this? I'll present the next section two ways, one with standard UNIX tools and one with ni, a tool I wrote for data science.

In both cases we're solving the same problem: let's build a list of articles sorted by the fraction of web citations, as opposed to other types like books or journals.

2.3 Wikipedia with standard UNIX tools

First we need a way to preview the data without decompressing the whole thing. The simplest strategy is to decompress into `less`:

```
$ bzip2 -dc enwiki* | less
<mediawiki xmlns="http://www.mediawiki.org/xml/export-0.10/"
           xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
           xsi:schemaLocation="http://www.mediawiki.org/xml/export-0.10/
                               http://www.mediawiki.org/xml/export-0.10.xsd"
           version="0.10" xml:lang="en">
  <siteinfo>
    <sitename>Wikipedia</sitename>
    <dbname>enwiki</dbname>
    <base>https://en.wikipedia.org/wiki/Main_Page</base>
    <generator>MediaWiki 1.30.0-wmf.14</generator>
    <case>first-letter</case>
...
```

Paging around a bit, the basic structure looks roughly like this:

```
<page>
  <title>AccessibleComputing</title>
  <ns>0</ns>
  <id>10</id>
  <redirect title="Computer accessibility" />
  <revision>
    <id>767284433</id>
    <parentid>631144794</parentid>
    <timestamp>2017-02-25T00:30:28Z</timestamp>
    <contributor>
      <username>Godsy</username>
      <id>23257138</id>
    </contributor>
    <comment>[[Template:This is a redirect]] has been deprecated, change to [[Template:Redi
    <model>wikitext</model>
    <format>text/x-wiki</format>
    <text xml:space="preserve">#REDIRECT [[Computer accessibility]]

{{Redirect category shell|
{{R from move}}
{{R from CamelCase}}
{{R unprintworthy}}
}}</text>
    <sha1>ds1crfrjsn7xv73djcs4e4aq9niwanx</sha1>
  </revision>
</page>
```

Inline citations are in the text and look like this (normally one line; I've reformatted here):

```
{{cite book
  |last=Dielo Trouda |authorlink=Dielo Truda
  |title=Organizational Platform of the General Union of Anarchists (Draft)
  |origyear=1926 |url=http://www.anarkismo.net/newswire.php?story_id=1000
  |accessdate=24 October 2006 |year=2006 |publisher=FdCA |location=Italy
  |archiveurl= https://web.archive.org/web/20070311013533/http://www.anarkismo.net/newswire
  |archivedate= 11 March 2007&lt;!--Added by DASHBot--&gt;}}
```

Overall we have two stages to this process. The first should convert the XML stream to a series of rows, let's say TSV of `webcount othercount title`, one per article. So the `AccessibleComputing` not-really-article above would look like `0 0 AccessibleComputing`.

```
# wikipedia-cite-extract.pl: count citations by type per article
use strict;
use warnings;
while (<STDIN>)
{
  # Skip rows until we hit a title, which will be stored in $1
  next unless /<title>(.*)<\/title>/;

  # Save the title and read until the end of the article text, counting any
  # citations we find.
  my $title = $1;

  for (my ($web, $other) = (0, 0);
       !eof and ($_ =~ <STDIN>) !~ /<\/text/;)
  {
    /^web$/ ? ++$web : ++$other for /\{\{cite (\w+)/g;
  }
  print join("\t", $web, $other, $title), "\n";
}
```

This runs one line at a time and streams its output, so we can quickly preview/debug/iterate. Here's what that looks like:

```
$ bzcat enwiki* | perl wikipedia-cite-extract.pl | less
0       0       AccessibleComputing
57      74      Anarchism
0       0       AfghanistanHistory
0       0       AfghanistanGeography
0       0       AfghanistanPeople
0       0       AfghanistanCommunications
0       0       AfghanistanTransportations
0       0       AfghanistanMilitary
0       0       AfghanistanTransnationalIssues
0       0       AssistiveTechnology
0       0       AmoeboidTaxa
18      207     Autism
0       0       AlbaniaHistory
...
```

OK, we want a ratio, so let's remove citation-free articles and calculate web/total as a fraction:

```
$ bzcat enwiki* \
    | perl wikipedia-cite-extract.pl \
    | perl -ane 'print join("\t", $F[0] / ($F[0] + $F[1]), @F[2..$#F]), "\n"
                   if $F[0] + $F[1]' \
    | less

0.435114503816794       Anarchism
0.08    Autism
0.566666666666667       Albedo
0.272727272727273       A
0.826086956521739       Alabama
0.181818181818182       Achilles
0.123529411764706       Abraham Lincoln
0.147540983606557       Aristotle
...
```

...and finally, before we write anything, let's sort the list by the fraction using `sort`. Before I do that, though, I want to talk a little about how `sort` works.

If you're sorting a stream of things, you have to store the whole stream first. Then, once you have everything, you shuffle stuff $\log n$ times and emit the sorted values.

This requires $O(n)$ space, of course, which is inconvenient: you now have at least one temporary copy of the data you're sorting. If you were running this in a language like Python, Perl, or Ruby this would all happen in memory, which limits the size of data you can sort using standard APIs. UNIX `sort` is different, though.

Internally, `sort` keeps only a very small amount of data in memory, by default something like 4MB at a time. Once it hits that limit, it writes the sorted buffer to a temporary file and sorts the next one, later merging them back from disk. GNU `sort` in particular supports some extra options that are useful for data like this: `--compress-program` and `--parallel`. `--compress-program` instructs `sort` to compress its temporary files, effectively reducing the *disk* space-complexity of the sort to $O(k)$, where k is the compressed size of your data. This, obviously, can make a huge difference.

So, with that said, here's the final pipeline:

```
$ sudo apt install pv pbzip2
$ pv enwiki* \
    | pbzip2 -dc \
    | perl wikipedia-cite-extract.pl \
    | perl -ane 'print join("\t", $F[0] / ($F[0] + $F[1]), @F[2..$#F]), "\n"
                   if $F[0] + $F[1]' \
    | sort -rn --compress-program=gzip \
    | gzip > wiki-sorted.gz
```

`pv` is `cat`, but with a progress meter that looks like this:

```
 164MiB 0:00:31 [5.39MiB/s] [>                                  ]  1% ETA 0:42:08
```

Having progress meters is crucial for long-running data jobs, if for no other reason than to make sure it looks reasonable.

While that's running and before I get to the `ni` version, let's talk about some tools useful for performance monitoring.

2.4 Monitoring tools

I have a standard set of things I install on Linux boxes that includes:

- `htop`: `top`, but better
- `atop`: `top` for CPU, memory, disk, network, etc
- `units`: a unit-aware calculator (e.g. `50GB/5Mbps in hours`)

TODO: make this section less sad and lonely

2.5 Wikipedia with `ni`

```
$ sudo apt install git pbzip2 perl perl-modules
$ git clone git://github.com/spencertipping/ni
$ sudo ln -s $PWD/ni/ni /usr/bin/
```

`ni` will preview compressed data automatically, so you can use it like a compression-aware `less` by default. It also knows to use `pbzip2` if you have it installed.

```
$ ni enwiki*          # preview data
$ ni enwiki* r/cite/  # select rows matching the regex /cite/, preview those
```

There's extensive documentation on how `ni` works, which may be helpful to understand what these commands do.

`wikipedia-cite-extract.pl` and the following perl command can be folded into a four-liner, and `sort -rn | gzip` becomes `oz`:

```
$ ni enwiki* p'return () unless /<title>(.*)<\/title>/;
               my $t  = $1;
               my @cs = map /\{\{cite (\w+)/, ru {/<\/text>/} or return ();
               r grep(/^web$/, @cs) / @cs, $t' oz > wiki-sorted.gz
```

`ni` monitors the data progress for you, so you can see a preview of the data moving out of each pipeline stage as well as speed and bottleneck pressure.

Whether you use `ni`, `perl`, or something else, command-line data processing is crucial to fast iteration on datasets. Compared to Hadoop/Spark, it's far less typing and effort, instant startup and debugging, and no data movement (and often faster; I'll cover that in more detail in later chapters).

2.6 Homework

1. What is the tenth most common word in Wikipedia? Assume you're running in a memory-constrained environment.

2. Write a simple disk-backed `sort` utility in your favorite language.

3. Given the TSV of `web other title` we built above, what is the relative overhead of parsing integers (vs line processing + tab splitting) when we filter the data? You could answer this question for `perl` or any other scripting language.

4. At what point does `pv` become the bottleneck in a pipeline? Is `cat` or `dd` faster? What's the most important implementation difference that makes them perform differently?

5. Use the `split` core utility to break Wikipedia into smaller files, without writing any uncompressed data to disk. What is the fastest way to count all citations in these pieces?[10] What are the tradeoffs that govern how large these pieces should be?

6. `xargs` shares the output file descriptor across its child processes, which can lead to data corruption if you use it in conjunction with `-P`. Write a pipeline that has this problem.

7. Given that Perl is ultimately using the `read()` system call, what machinery is involved to implement the "read a line from `stdin`" operation?

[10]Hint: `xargs` may be useful if you have multiple processors.

3 Canvas

OSM Planet is one of my favorite datasets, if for no other reason than the fact that it's so much fun to visualize. Like Wikipedia, it's also stored in an awkward XML/bzip2 format; unlike Wikipedia, though, this format has some relational elements that complicate processing a bit unless you have tons of memory. First things first: let's get it downloaded (this requires about 70GB of free space):

```
$ sudo apt install wget
$ wget https://planet.osm.org/planet/planet-latest.osm.bz2
```

While that's running, here's some pontification about how awful it is to be a data scientist.

3.1 Visualizing stuff

As a rule, data is too damn big. Not for any specific purpose, it's just too big in general. There are some common options for visualizing things:

- MatPlotLib
- Bokeh
- GnuPlot
- D3
- `ni --js`[11]

My main beef with most existing solutions, though, is that they prioritize presentation over scale. Let's take gnuplot as an example: it uses a custom rendering system, so no DOM objects (good), but it's not really cut out to handle a million points. In my experience it takes the view a good 20-30 seconds to pan/zoom/etc at that scale.

The problem with this, of course, is that a million isn't "big data" or anywhere close. In any ordinary problem, one million low-dimensional entities (i.e. not entire articles or something) would be considered *really small* data. To get a sense of scale, let's count some entities from various datasets:[12] [13]

[11] Full disclosure: I wrote this one, but there's a reason I added it to this list

[12] Each of these commands is almost entirely decompression-bound; this is a great example of a use case where LZ4 or similar comes in handy. These jobs would be faster if I had sharded out the files up front, but not by much: the 12-disk RAID6 is about 60% IO-bound as it is and the storage server only has eight processors.

[13] These jobs consumed about 180W over baseline and ran for a couple of hours, so the overall cost was $0.061 – or $0.02/TB. I realize it's not an entirely fair comparison, but this is 250x cheaper than the $5/TB that Google BigQuery charges.

```
# Wikipedia: how many articles? (xz/ubuntu: 45MB/s)
$ ni enwiki-7820 S24r'/<page>/' e'wc -l'
17,773,690

# OpenStreetMap: how many vertices? (LZ4/gentoo: 480MB/s)
$ ni osm-planet-2018.0101.lz4 S8e[grep -c '<node'] ,sr+1
4,254,954,891

# NYC Taxicab: how many pickup/dropoffs? (LZ4/gentoo: 370MB/s)
$ ni nyctaxi-trip.lz4 e'wc -l'
173,179,759

# Reddit comments: how many comments from 2005 to 2017? (LZ4/ubuntu: 335MB/s)
$ ni RC_*.lz4 e'wc -l'
3,012,842,697
```

I should also point out that none of these datasets are "big data" in an industrial sense; they're just things I have lying around on the storage server. I think they all compress to under 1TB. But already we're dealing with billions of data points, which is obviously a lot more than most visualization tools are built to handle.

So where do things go sideways? Let's look at it in terms of bytes of heap usage per data point. I'm going to make some rough guesses based on how these tools work:

- MatPlotLib: uses NumPy for storage, so 4-8 bytes/object/dimension
- Bokeh: uses WebGL, so 8 bytes/object/dimension (probably? TODO)
- Gnuplot: native C code; it looks like 64 bytes/object with struct padding
- D3: backs into SVG, so at least 100 bytes/object (most likely 200-400)
- `ni --js`: uses `Float64Array` for storage and reservoir sampling, so ≤ 8 bytes/object/dimension

I don't want to rag on D3 and Gnuplot too much; these are both great tools when your data is small enough. But anytime you're starting with a billion data points and your visualization layer is only good to 10K, that's another thing you'll end up needing to manage – and that drives up your costs/iteration time.