The Basic Problem

Someone hands you a large data set. How do you start analyzing it?

In particular:

- how do you get a first impression of the data?
- how do you pick the most appropriate tool for the task?
- how do you you decide about the required hardware?

File Formats

Data sets can be exchanged using a huge number of formats.
Some popular choices:

- text files / CSV files
- XML
- JSON
- binary but open formats like RCFile/OrcFile/Parquet
- proprietary binary formats (not well suited for exchange)

When in doubt, `file` can be a useful tool

CSV Files

A plain text file contains rows of data (Comma Separated values):

```
Sherlock Holmes,221B Baker Street,Detective
James Moriarty,Reichenbach Falls,Villain
```

- simple, human readable formats
- very popular, widely used
- but a lot of subtleties
- separator customizable
- strings with separator require quoting
- `"String with "" inside"`

XML Files

A text format encoding (semi-)structured data:

```
<characters>
<person name="Sherlock Holmes" address="221B Baker Street"
job="Detective"/>
<person name="James Moriarty" address="Reichenbach Falls"
job="Villain"/>
</characters>
```

- format itself better standardized than CSV
- suitable for nested data, too
- human readable, but not very friendly to write
- very verbose, also full XML spec is very complex
- allows for advanced features (XML Schema etc.), but these are rarely used
- somewhat popular, but declining

JSON Files

A text format stemming from JavaScript Object Notation:

```
[{"name":"Sherlock Holmes", "address":"221B Baker Street",
"job":"Detective"},
{"name":"James Moriarty", "address":"Reichenbach Falls",
"job":"Villain"}]
```

- use cases similar to XML
- but a much simpler format
- less verbose and easier to write
- growing popularity

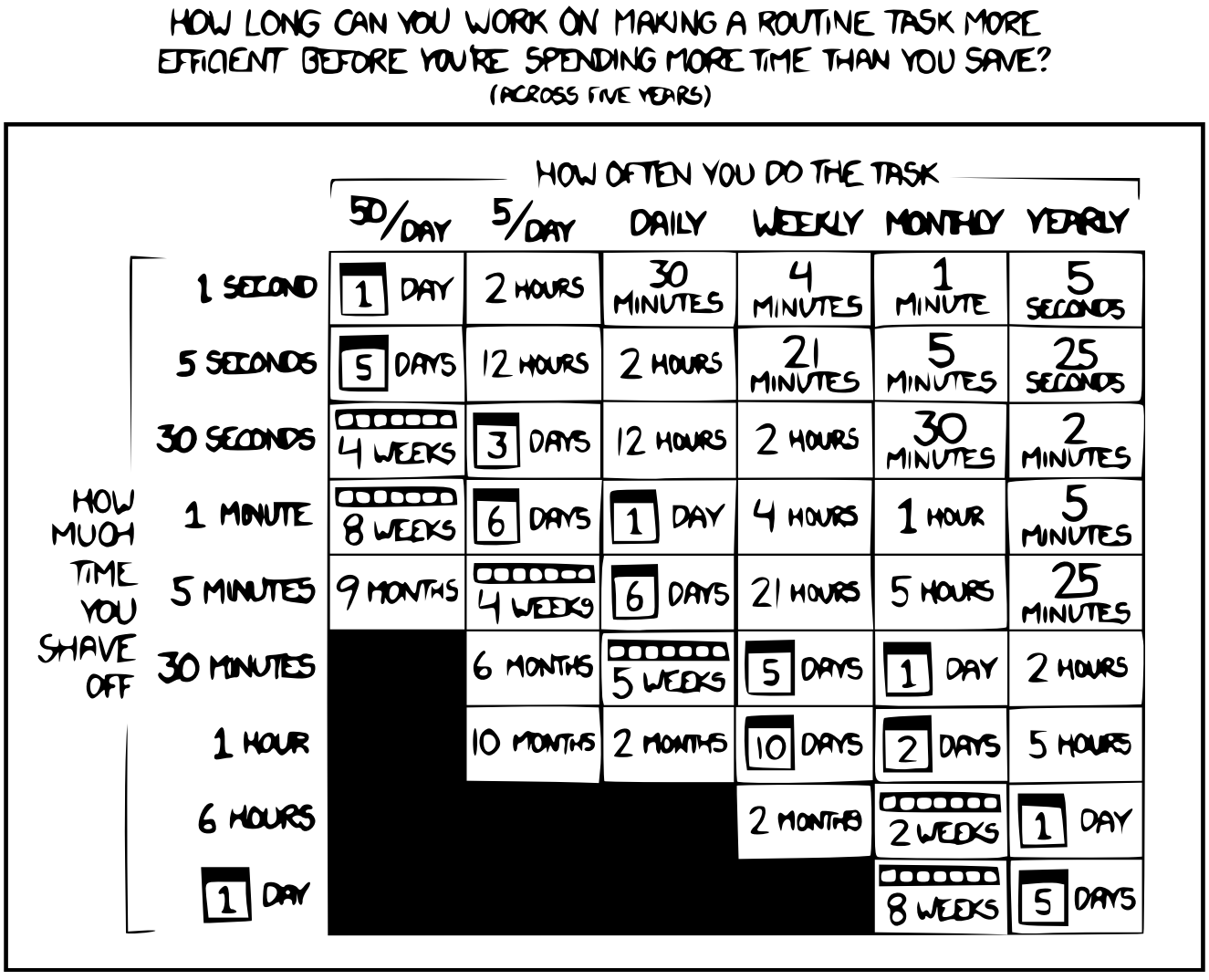
Working With Text Files

Text files are quite popular because they are human readable and easy to use.

We start with **command line tools**

- available by default on Unix/Linux platforms
- simple, but surprisingly **powerful in combination**
- ideal to get **a first impression**
- allows to examine and explore the data
- ultimately we want to pick a better tool
- but command line tools allow for simple analysis without any code

Is It Worth the Time?



http://xkcd.com/1205/ CC BY-NC 2.5

Combining Tools

Many tools can read and write file, but it is often more convenient to combine them with pipes

- every program has an input stream and an output stream
- concatenating both:
  `command1 | command2`
- redirecting to a file
  `command > file`
- redirecting from a file
  `command < file`
- can form long pipelines of commands
- in addition: error stream
  `command 2> /dev/null`

Reading Files - `cat`

- reading a single file
  `cat file`
- concatenating multiple file
  `cat file1 file2 file3`
- accepting piped input
  `command | cat`
- escaping binary data
  `cat -v binaryfile`
- for compressed files
  `zcat / bzcat / xzcat`
- mainly used as input for other commands

Paging Results - `less`

- used to inspect results or files
- in pipelines mainly used as last command
- paging a command result
  `command | less`
- paging a file
  `less file`
- chopping long lines
  `less -S`

Filtering - grep

Returns all qualifying lines

- filtering input
  ```
  cat file1 file2 | grep 'img[0-9]*\.jpg'
  ```
- regular expression syntax.
  Note: might need escapes in shell
- filtering files
  ```
  grep 'img[0-9]*\.jpg' *.html
  ```
- returning only the match
  ```
  command | grep -o 'user=.*$'
  ```
- returning only non-matching lines
  ```
  grep -v '^warning' logfile
  ```
- case insensitive search
  ```
  cat file | grep -i '\.jpg'
  ```

A very powerful tool with many options

Examples

Find all senders on the LKML

Find all senders that are not using .com domains

Extract the email part of the sender address

Sorting - `sort`

Sorts the input according to a criterion

- basic sorting
  ```
  cat file | sort
  ```
- sorting numerically
  ```
  cat file | sort -n
  ```
- sorting a specific key
  ```
  cat file | sort -k 2
  ```
- sorting with explicit separator
  ```
  cat file | sort -t ',' -k 3
  ```
- sorting, eliminating duplicates
  ```
  cat file | sort -u
  ```
- very powerful and useful
- handles files larger than main memory
- sorted files used as inputs for other algorithms

Selecting Prefix/Suffix - `head`/`tail`

Returns the begin/end of a file

- first 20 entries
  ```
  sort file | head -20
  ```
- last 15 entries
  ```
  sort file | tail -15
  ```
- everything but the first two entries
  ```
  sort file | tail -n +2
  ```
- everything but the last two entries
  ```
  sort file | head -n -2
  ```
- useful for the final result
- but also useful for min/max computations

Handling Duplicates - `uniq`

Handles duplicates in **sorted** input

- eliminating duplicates
  ```
  sort file | uniq
  ```
- counting duplicates
  ```
  sort file | uniq -c
  ```
- returning duplicates
  ```
  sort file | uniq -d
  ```
- returning unique lines
  ```
  sort file | uniq -u
  ```
- very useful for grouping and counting

Examples

Find all posters to the LKML, sorted alphabetically.

Find the 10 most prolific posters to the LKML.

Find all posters who wrote only a single mail.

Projecting Columns - cut

Allows to return only relevant parts of the input

- return specific fields
  ```
  cat file | cut -f 1,3
  ```
- cut with specific delimiter
  ```
  cat file | cut -f 2-4 -d ','
  ```
- using characters instead of fields
  ```
  cat file | cut -c 1-10
  ```

Counting - `wc`

Counts the number of lines/words/characters

- full statistics
  `cat file | wc`
- count the number of lines
  `cat file | wc -l`
- count the number of bytes
  `cat file | wc -c`
- useful to gather statistics about the data

Shuffling - `shuf`

Shuffles the lines of the input

- random order
  `cat file | shuf`
- not so commonly used, but very useful in some cases
- obtaining a random sample
  `cat file | shuf | head -10000`
- result much more useful than without `shuf`
- also for performance tests

Examples

Count the number of mail in the archive

Pick 20 mail subjects at random

Compute the most popular content type

Editing Text - sed

- replacing one text with another
  `cat file | sed 's/oldText/newText/'`
- replacing all occurrences of a text
  `cat file | sed 's/oldText/newText/g'`
- using back-references
  `cat file | sed 's/IMG_\([0-9]*\).JPG/image\1.jpg/g'`
- case insensitive matching
  `cat file | sed 's/file[0-9]*.png/"&"/I'`
- using multiple patterns
  `cat file | sed -e 's/old1/new1/' -e 's:a/b:a_b:'`
- extremely powerful and useful
- many more features (grouping, conditional branching, etc.), but these are rarely used

Combining Files - `join`

Combines sorted input files

- combining two files on a common field
  ```
  join -1 2 -2 1 file1 file2
  ```
- combining with explicit delimiter
  ```
  join -1 2 -2 1 -d ',' file1 file2
  ```
- preserving unmatched lines
  ```
  join -1 2 -2 1 -a 1 file1 file2
  ```
- behaves like a relational join
- but if you need that it might be better to use more powerful tools

Computations - awk

Executes a program for every line of input

- general structure
  ```
  awk 'BEGIN { init-code } { per-line-code } END { done-code }' file
  ```
- counting the number of lines
  ```
  awk 'BEGIN { x=0 } { x=x+1 } END { print x }'
  ```
- summing up a column
  ```
  awk 'BEGIN { x=0 } { x=x+$2 } END { print x }'
  ```
- averaging a column
  ```
  awk 'BEGIN { x=0; y=0 } { x=x+$2; y=y+1 } END { print x/y }'
  ```
- conditions
  ```
  awk 'BEGIN { x=0 } { if ($1> 10) x=x+$2 } END { print x }'
  ```
- and many other features

Examples

Find the most popular words in the LKML data set

Compute the average number of lines in a mail

Find all mail authors who never get a response

Understanding the Performance Spectrum

So far we have looked at high-level text tools for analysis.
Now we want to understand the performance spectrum

- what kind of performance can we expect from a single box?
- how large is the spread from high-level tools to hand-written code?
- what are the theoretical limits?
- what is the impact of careful programming?
- how large would a cluster have to be to improve performance?
- ultimately, we want to know which tools and hardware we need for a specific problem

To understand that, we take a simple problem and look at it in details, optimizing as much as needed.

The Toy Problem

As demonstrator, we take the TPC-H dataset, relation `lineitem` and sum up `quantity` (5th column):

```
1|155190|7706|1|17|21168.23|0.04|0.02|N|O|1996-03-13|...
1|67310|7311|2|36|45983.16|0.09|0.06|N|O|1996-04-12|...
1|63700|3701|3|8|13309.60|0.10|0.02|N|O|1996-01-29|...
1|2132|4633|4|28|28955.64|0.09|0.06|N|O|1996-04-21|...
...
```

- 725MB * SF (total data set roughly 1GB * SF)
- 6 million * SF lines
- we use text input for now
- real benchmark is much more complex, of course
- but that simple query can be analyzed easily and is surprisingly expensive

Performance Limits

What are the the limits for that query (SF1, 725MB)?

	bandwidth	query time
1GB ethernet	100MB/s	7.3s
rotating disk	200MB/s	3.6s
SATA SSD	500MB/s	1.6s
PCIe SSD	2GB/s	0.36s
DRAM	20GB/s	0.04s

- completely ignores CPU costs
- CPU costs not so relevant for disks, but very relevant for DRAM
- we will not be able to get that performance
- but we should try to get close

Computing the Sum - awk

Using awk the task is trivial to formulate

```
awk -F '|' 'BEGIN {x=0} {x=x+$5} END {print x}' lineitem
```

Computing the Sum - awk

Using awk the task is trivial to formulate

```
awk -F '|' 'BEGIN {x=0} {x=x+$5} END {print x}' lineitem
```

- first execution: 4.5s
- second execution: 3.6s
- first execution was waiting for disk
- second execution was CPU bound (much slower than DRAM speed)
- main memory/caching has a huge effect

Computing the Sum - Python

We can write a small Python program to compute the sum

```
sum=0
with open(sys.argv[1]) as f:
 for line in f:
  sum=sum+float(line.split('|')[4])
print (sum)
```

Computing the Sum - Python

We can write a small Python program to compute the sum

```
sum=0
with open(sys.argv[1]) as f:
 for line in f:
  sum=sum+float(line.split('|')[4])
print (sum)
```

- first execution: 6.2s
- second execution: 6.2s
- Python is always CPU bound!
- Cannot keep up even with a rotating disk

Computing the Sum - C++

We can do the same in C++

```
unsigned sum=0;
while (getline(in,s)) {
 unsigned v=0;
 for (auto iter=s.begin(),limit=s.end();iter!=limit;++iter)
  { ...  } /* extract 5th column into v */
 sum+=v;
} cerr << sum << endl;
```

Computing the Sum - C++

We can do the same in C++

```
unsigned sum=0;
while (getline(in,s)) {
 unsigned v=0;
 for (auto iter=s.begin(),limit=s.end();iter!=limit;++iter)
  { ...  } /* extract 5th column into v */
 sum+=v;
} cerr << sum << endl;
```

- first execution: 3.5s
- second execution: 0.8s
- first execution is I/O bound, second is CPU bound
- much faster than the others, but still far from DRAM speed
- code is more complex but also more efficient

Where Does the Time Go?

Figuring out where a program spends time requires profiling:

```
perf record sum lineitem && perf report
```

Counter summaries are interesting, too:

```
perf stat -d sum lineitem
```

Computing the Sum - C++ with mmap

We can avoid copying the data by mapping the file into memory

```
int handle=open(argv[1],O_RDONLY);
lseek(handle,0,SEEK_END);
auto size=lseek(handle,0,SEEK_CUR);
auto
data=mmap(nullptr,size,PROT_READONLY,MAP_SHARED,handle,0);
...  /* operates directly on data, without manual reads */
```

Computing the Sum - C++ with mmap

We can avoid copying the data by mapping the file into memory

```
int handle=open(argv[1],O_RDONLY);
lseek(handle,0,SEEK_END);
auto size=lseek(handle,0,SEEK_CUR);
auto
data=mmap(nullptr,size,PROT_READONLY,MAP_SHARED,handle,0);
...  /* operates directly on data, without manual reads */
```

- first execution: 3.7s
- second execution: 0.65s
- cold cache slightly slower, but warm cache faster
- we can directly read the file system cache, without copies
- profile has changed

Computing the Sum - blockwise terminator search

We can speed up searching for \n by using block operations

```
uint64_t block=*reinterpret_cast<const uint64_t*>(iter);
constexpr uint64_t high=0x8080808080808080ull;
constexpr uint64_t low=0x7F7F7F7F7F7F7F7Full;
constexpr uint64_t pattern=0x0A0A0A0A0A0A0A0Aull;
uint64_t lowChars=(~block)&high;
uint64_t found0A=~((((block&low)^pattern)+low)&high);
uint64_t matches=matches0A&lowChars;
```

Computing the Sum - blockwise terminator search

We can speed up searching for \n by using block operations

```
uint64_t block=*reinterpret_cast<const uint64_t*>(iter);
constexpr uint64_t high=0x8080808080808080ull;
constexpr uint64_t low=0x7F7F7F7F7F7F7F7Full;
constexpr uint64_t pattern=0x0A0A0A0A0A0A0A0Aull;
uint64_t lowChars=(~block)&high;
uint64_t found0A=~((((block&low)^pattern)+low)&high);
uint64_t matches=matches0A&lowChars;
```

- first execution: 3.7s
- second execution: 0.45s
- no effect on cold cache time, but warm cache significantly faster
- using SSE instructions even faster (0.41s), but not portable

Computing the Sum - parallelism

We can use multiple cores for processing

```
unsigned chunks=thread::hardware_concurrency();
vector<thread>threads;
for (unsigned index=1;index<chunks;++index)
  threads.push_back(thread(sumChunk(index));
sumChunk(0);
```

Computing the Sum - parallelism

We can use multiple cores for processing

```
unsigned chunks=thread::hardware_concurrency();
vector<thread>threads;
for (unsigned index=1;index<chunks;++index)
  threads.push_back(thread(sumChunk(index));
sumChunk(0);
```

- first execution: 4.3s
- second execution: 0.13s
- warm cache case parallelizes nicely, but cold cache get slower!
- due to random access on a rotating disk
- we could squeeze out some more performance, but that is roughly the limit when using only simple tricks

Comparing the Performance

Warm cache times:

	time	bandwidth
awk	3.6*s*	201 MB/s
Python	6.2*s*	116 MB/s
C++	0.8*s*	906 MB/s
+ mmap	0.65*s*	1,115 MB/s
+ block	0.45*s*	1,611 MB/s
+ parallel	0.13*s*	5,576 MB/s

- implementation matters a lot
- even a desktop box can do multiple GB/s
- here we saturate everything but DRAM

Should we use a Cluster?

Depends on the situation

- if we have to **ship data**
  - network bandwidth 100MB/s (1GB/s for 10GB ethernet)
  - program bandwith 5GB/s
  - **no**, using a cluster slows down the computation
- if we can **ship computation**
  - performance determined by latency and number of nodes
  - roughly $20ms + \frac{|W|}{5GB/s*n}$
  - for 10 nodes pays off if $|W| \geq 113MB$
  - so **yes** if the problem is large enough to warrant the latency

Should we use a Different Representation?

Not always an option, of course

- but a columnar binary format would reduce touched data by factor 30
- plus we avoid the expensive parse logic
- often improves runtime by factor 50 or more
- will require a conversion step here
- but pays off for repeated queries

What We Have Seen

- **quality of implementation** matters a lot
- a single box can handle many gigabytes efficiently
- network speed is a problem / using remote CPUs is difficult
- we often prefer **scale up** over **scale out**
- not always possible/economical, of course
- scale out has significant overhead, pays off only in the long run