Pipes, Hash & Trees

- or -

What I've been doing over the last couple of months

Information Retrieval

- Retrieve information that is relevant to you from somewhere you couldn’t exhaustively look

- Most familiar context: Google

ATIRE

- In-house indexer & search engine in active development by Andrew & members of the IR lab since 2008
- Gaining popularity, actively used here and gaining traction overseas, particularly Australia
- 100K lines of C++

Indexing

- To retrieve potential documents quickly we create an index of terms to documents

- Almost exactly the same as an index found in books — but exhaustive and created automatically

Nitro

- [ ] 4x AMD Opteron 6276 2.3GHz 16-core
- [ ] 512GB memory
- [ ] 6x 600GB 10,000 RPM drives

The Pipeline

The Pipeline

- Indexing in ATIRE uses a multi-stage parallel producer/consumer inspired pipeline

- That's a lot of buzzword-y stuff!

The Pipeline

- Each section of the pipeline knows how to do its own thing
  - Un-gzip, Un-tar etc.
- Each section also knows how to request more data and respond to requests

The Pipeline

- [ ] Where/why are we waiting?
- [ ] Nowhere unexpected
  - [ ] Disk I/O
  - [ ] CPU intensive (un-gz-ing)

The Pipeline

- [ ] Add buffering
- [ ] When upstream asks for data, ask for more from downstream, so that when upstream asks again we can immediately respond
- [ ] Useful when dealing with disks in particular

The Pipeline

- [ ] Add double buffering
- [ ] While upstream is busy working, we can fill the remainder of our buffer
- [ ] Saturate the pipeline so nothing waits unnecessarily

The Pipeline

- [ ] Parallel indexing — pre-index documents separately and fold into the final index afterwards

- [ ] Non-parallel indexing — each document is indexed in turn directly into the final index

Benefits of a Lock-Free Tree

- ☐ Parallel indexed documents look for their nodes in the main hash table as they are created

- ☐ Because it didn't exist, doesn’t mean we can blindly create — node might have created by another document in the meantime

The Hash Function & Table

The Hash Table

- Each term is hashed and inserted into a binary search tree at the given hash-bucket
- What constitutes a good hash function?
  - General answer: uniform distribution of keys
  - IR answer: good distribution of keys

The Hash Functions

- Pearson's Fast Hash function:
  - Random walk of the string
- Header Hash I — developed internally
  - Treat head of string as a base-37 digit
  - Special case numbers to their value
  - High bits set to low bits of length to further distribute

The Hash Functions

- ☐ Header Hash II — developed internally
  - ☐ Special case numbers to their value
  - ☐ Treat head of string as a base-27 digit
  - ☐ Calculated in a different order, so small strings are closer together
  - ☐ No length component
  - ☐ WIP

The Hash Functions

- [ ] Header Hash III — developed internally
- [ ] Treat head of string as a base-27 digit
- [ ] Frequencies of characters lets us combine buckets, letting frequent longer terms be not collided with
- [ ] WIP

The Hash Table II

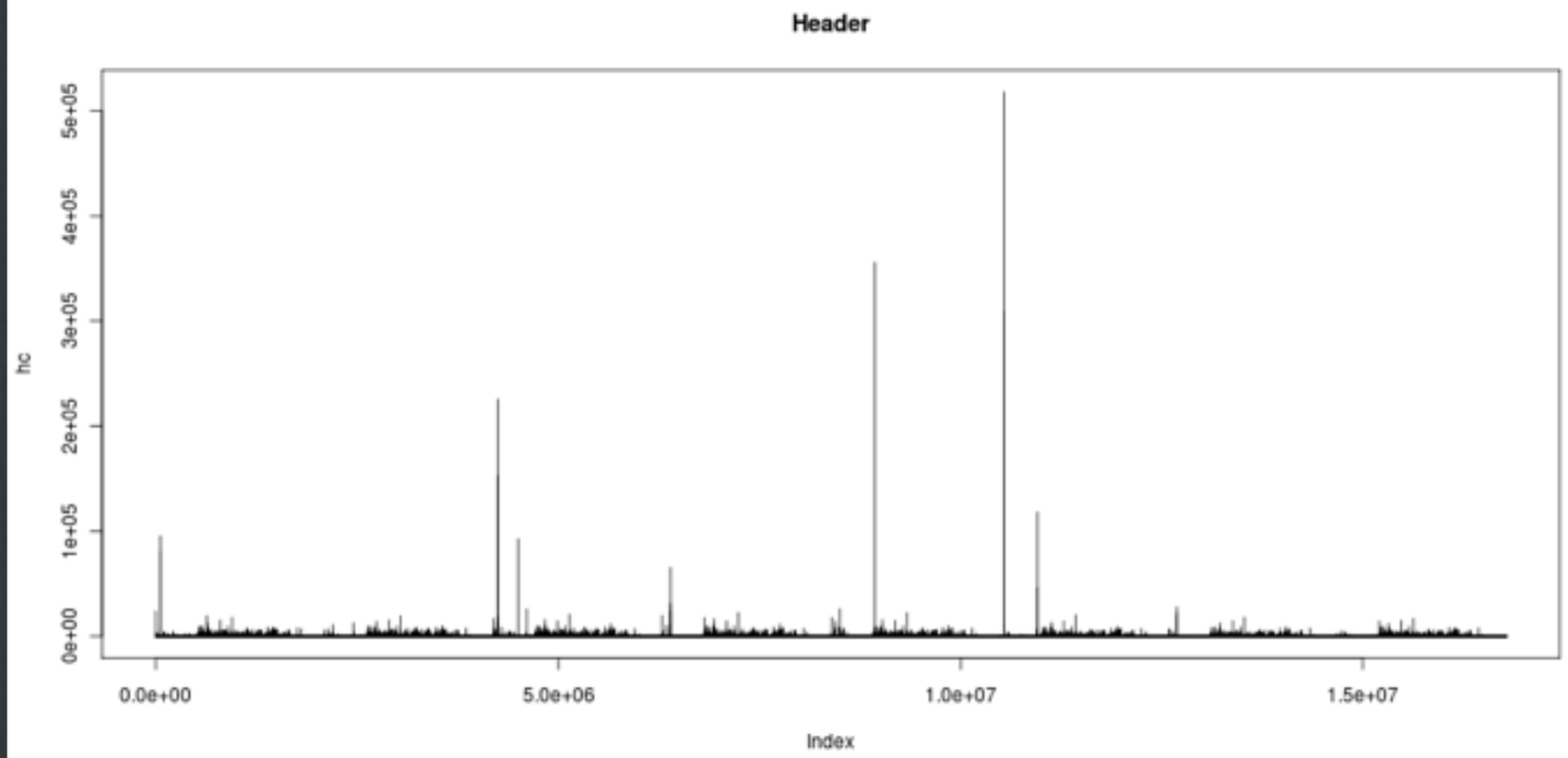
- How big do we make the hash table?

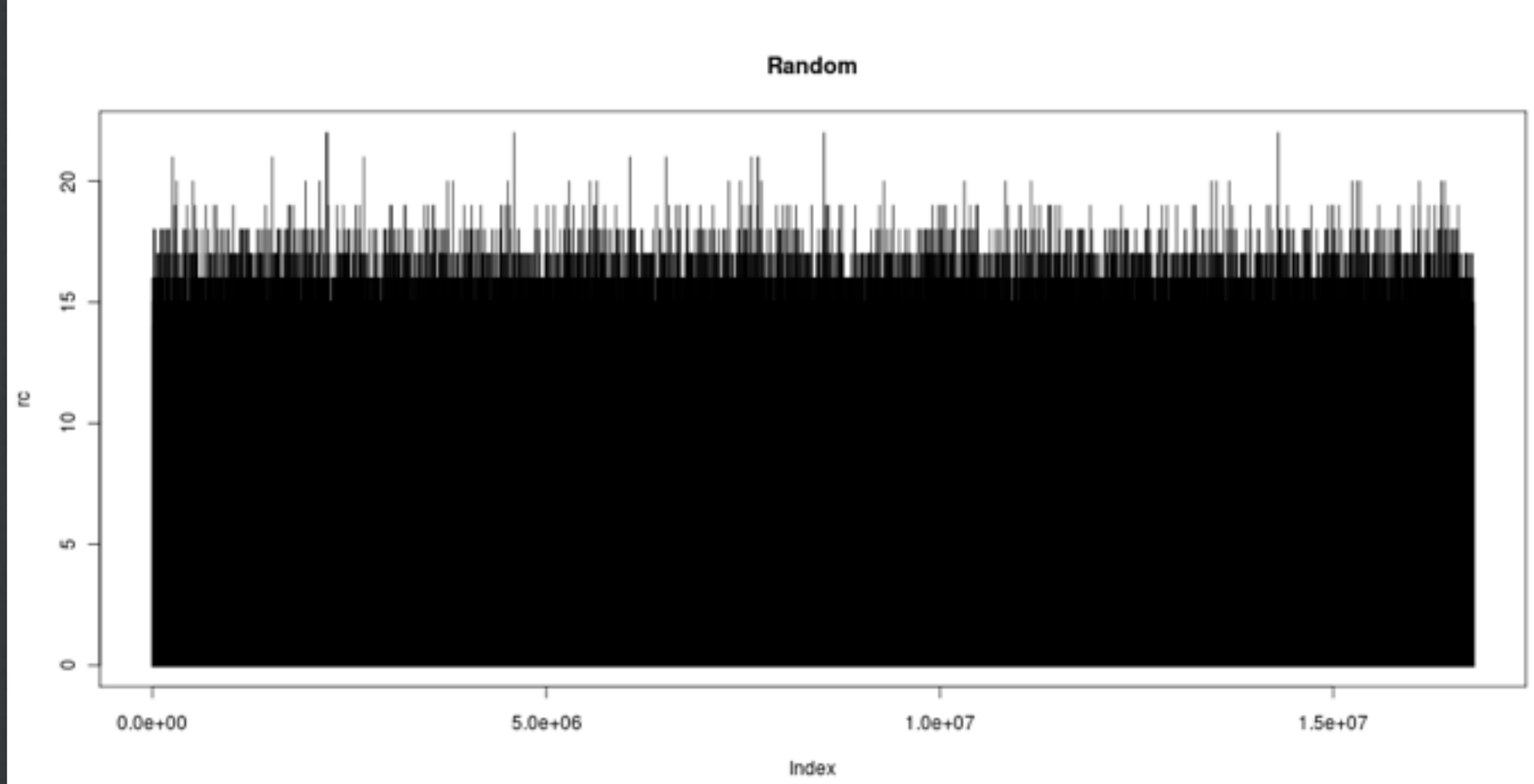
- Two hash tables — one per document, one overall

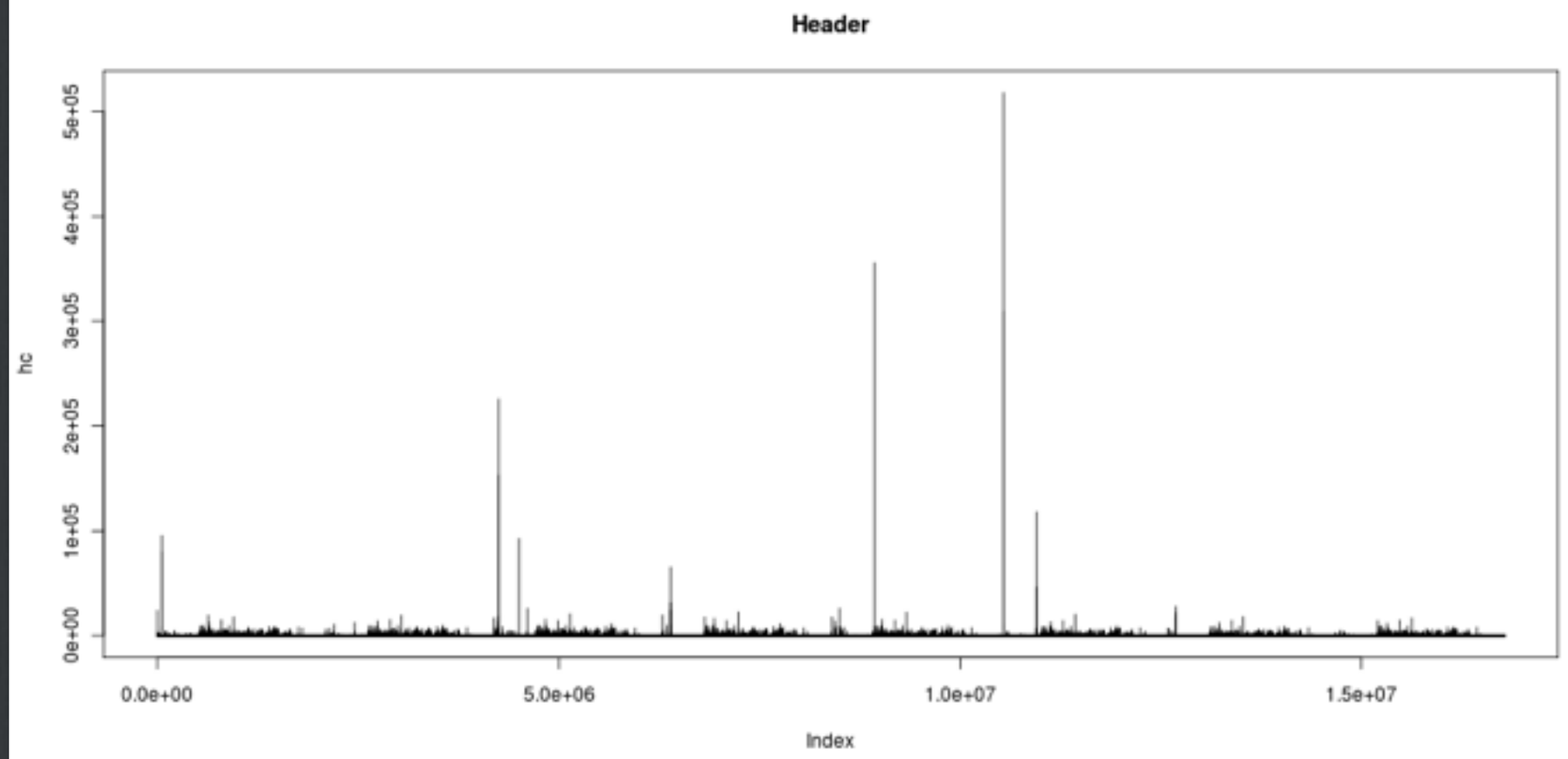
- Per document hash table size 256 — unlikely to be many intra-document hash collisions regardless of function

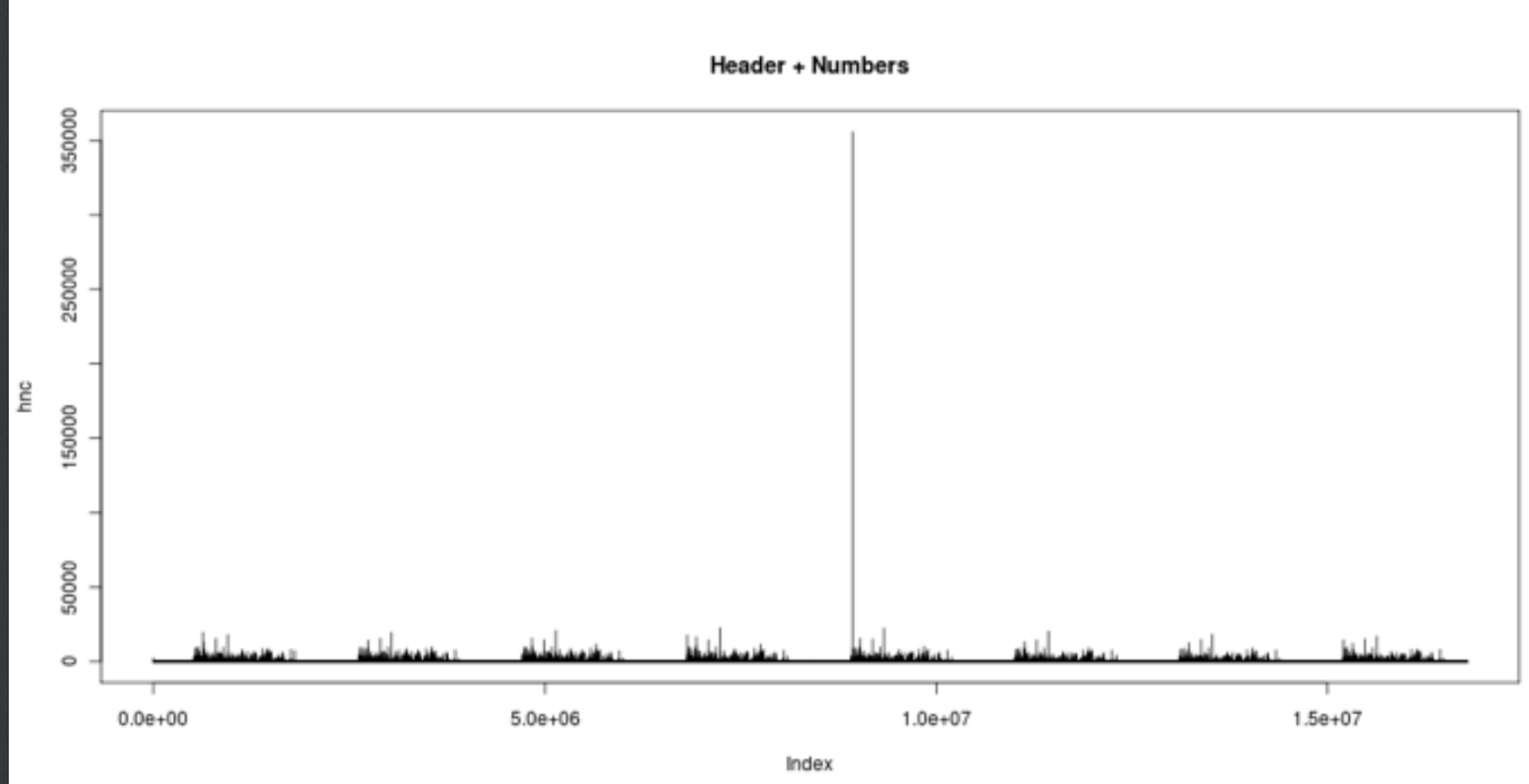
The Hash Table II

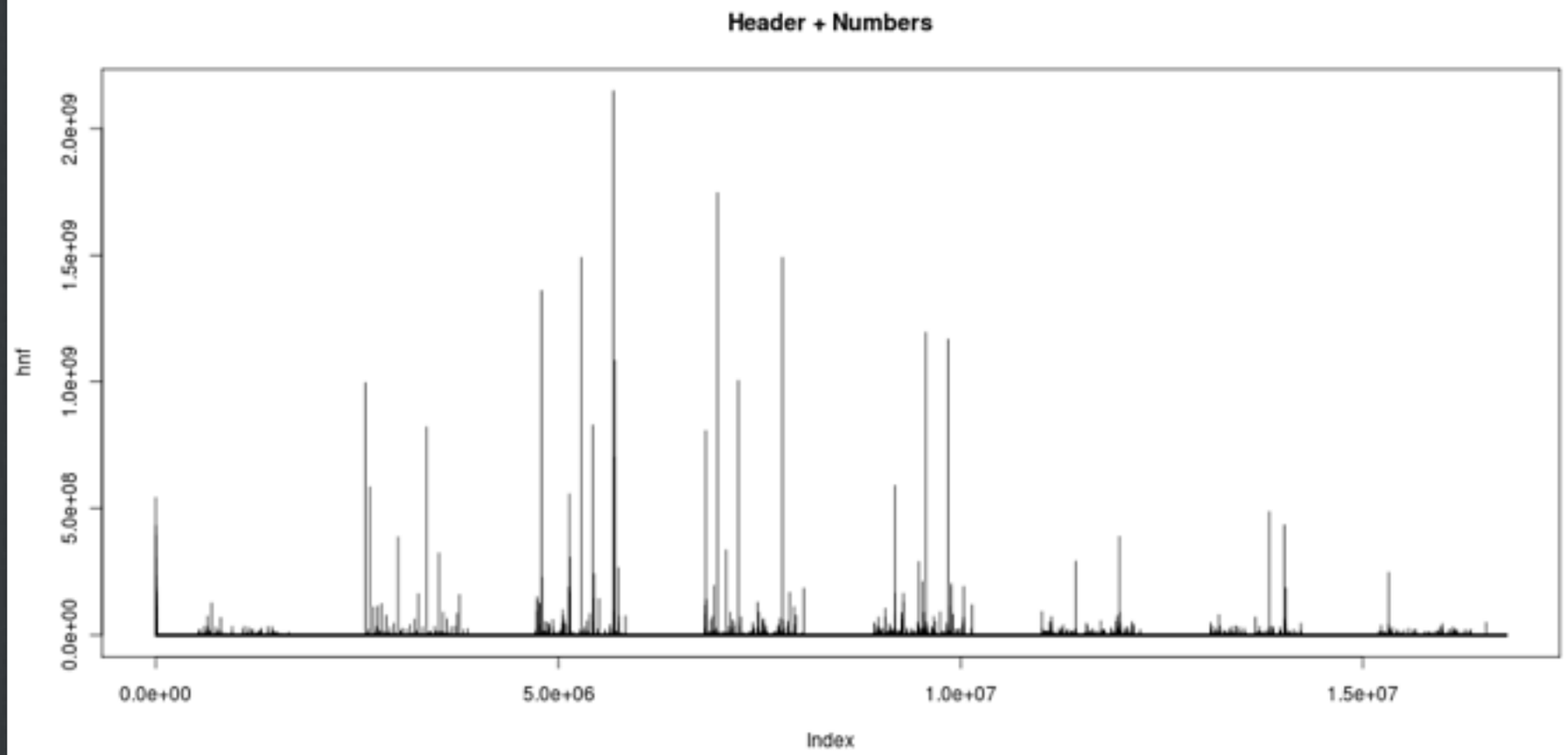
- [ ] 2**8 — obviously (?) not large enough
- [ ] 2**16 — too many collisions
- [ ] 2**24 — large enough to minimise collisions, small enough to allocate
- [ ] 2**32 — impractical to allocate on all but the biggest machines, but would allow perfect hashing!

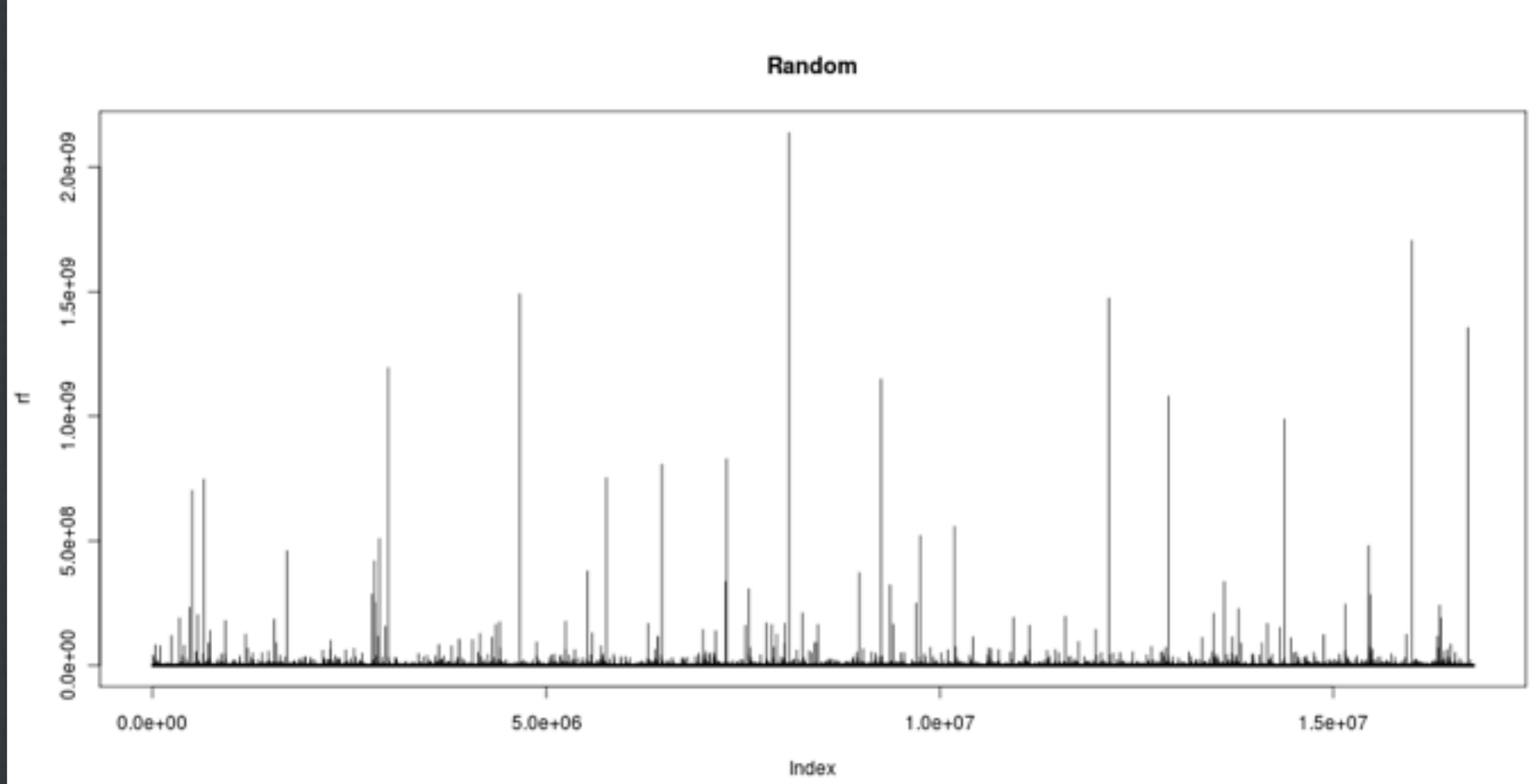

Header
hc
0e+00
1e+05
2e+05
3e+05
4e+05
5e+05
0.0e+00
5.0e+06
1.0e+07
1.5e+07
Index


Random
rc
0
5
10
15
20
0.0e+00
5.0e+06
1.0e+07
1.5e+07
Index


Header
hc
0e+00
1e+05
2e+05
3e+05
4e+05
5e+05
0.0e+00
5.0e+06
1.0e+07
1.5e+07
Index


Header + Numbers
hnc
0
50000
150000
250000
350000
0.0e+00
5.0e+06
1.0e+07
1.5e+07
Index


Header + Numbers
hnf
0.0e+00
5.0e+08
1.0e+09
1.5e+09
2.0e+09
0.0e+00
5.0e+06
1.0e+07
1.5e+07
Index


Random
rf
0.0e+00
5.0e+08
1.0e+09
1.5e+09
2.0e+09
0.0e+00
5.0e+06
1.0e+07
1.5e+07
Index

Which Hash Function

- Expensive part: string comparisons in the tree at each bucket
- If there are fewer unique terms than buckets, hash uniformly and every term gets a single strcmp
- If there are more unique terms than buckets, cheap access for frequently occurring terms is a must
- We assume the latter

Which Hash Function

- Non-parallel indexing:
  - Every term is inserted/updated into the global hash/ tree per occurrence
  - This depends on number of unique terms:
    < buckets — even distribution
    > buckets — skewed distribution

Which Hash Function

- Parallel indexing:
  - Each document likely has few unique terms, so even distribution in per-document hash-table
  - For merging into global distribution, terms that occur in lots of documents need to be cheap to access

The Hash Function

- [ ] Header hash, or variant (WIP)
- [ ] Works well for the majority of documents
- [ ] When collisions occur, the input is sufficiently random that the tree at each bucket is balanced

The Wild Wild Web

- Documents on the web are not typical by any definition of the word

- Even so, atypical documents are usually not a problem until…

The Wild Wild Web

- [ ] … we come across a long list of DNA sequences
- [ ] … that start with the same 4 characters
- [ ] … are the same length
- [ ] … and have been pre-sorted
- [ ] … and there are multiple documents like this

The Wild Wild Web

- ☐ BST at that hash-node degrades to a linked-list
- ☐ Traverse the linked list to insert the next term
- ☐ > 600k items … blows the stack when recursed down

Trees

Balancing Trees

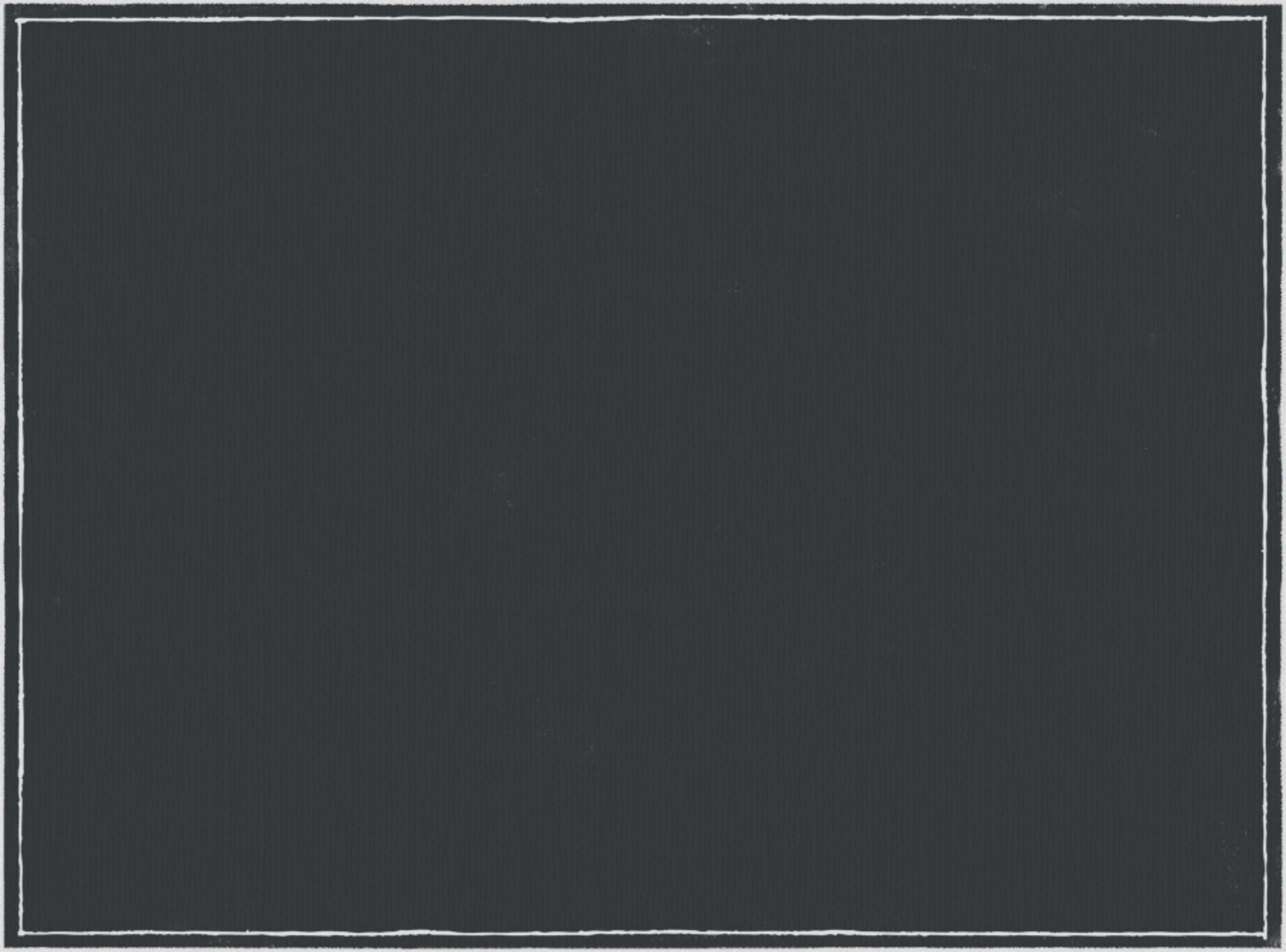
- Self-balancing trees (AVL-trees, red/black trees) need extra data (height/colour/parent) to be stored/calculated

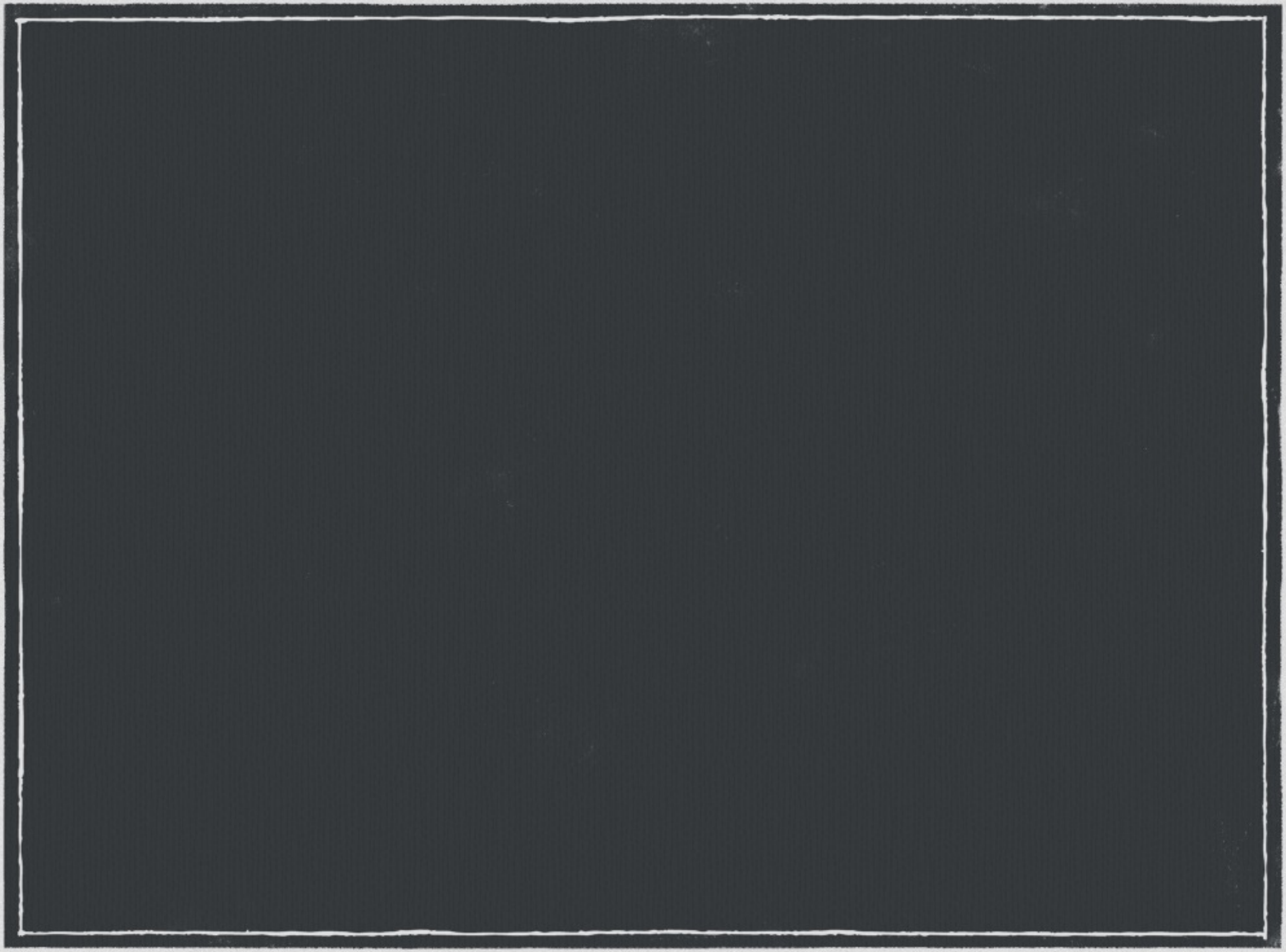
- Already seen it's only rare occasions we need to worry about the balance

- So one-off balancing, when required

Balancing Trees

- ☐ Day-Stout-Warren
- ☐ Rotate to degrade to a linked-list
- ☐ Perform series of rotations to generate a perfectly balanced tree afterwards
- ☐ Linear time, in-place, amortized cost





Balancing Trees

- ☐ Too frequently and spend too much time balancing
- ☐ Too infrequently and spend too much time inserting
- ☐ Never and we blow the stack and crash

Balancing Trees

- [ ] Merge from individual “index” to global is pre-order
- [ ] Merging balanced trees into the final index creates balanced trees

Conclusions

Conclusions

- Trade-offs and compromises occur all over the place
- What helps one problem creates others
  - eg: random hashing would solve DNA sequence blowing the stack, but be slower overall

Conclusions

- In theory, theoretically better always is
- In practice, it isn’t necessarily practical
- In general, generality can hinder performance

Conclusions

- As far as we are aware, nobody has created a hash function that purposefully generates an uneven distribution

Questions & Comments