Recap: lecture 2
- Stemming, tokenization etc.
- Faster postings merges
- Phrase queries

This lecture
- Index compression
- Space estimation

Corpus size for estimates
- Consider \( n = 1\text{M} \) documents, each with about \( L=1\text{K} \) terms.
- Avg 6 bytes/term incl spaces/punctuation
- 6GB of data.
- Say there are \( m = 500\text{K} \) distinct terms among these.

Don’t build the matrix
- 500K x 1M matrix has half-a-trillion 0’s and 1’s.
  - But it has no more than one billion 1’s.
  - Matrix is extremely sparse.
- So we devised the inverted index
  - Devised query processing for it
- Where do we pay in storage?
Storage analysis

- First will consider space for postings pointers
- Basic Boolean index only
  - Devise compression schemes
- Then will do the same for dictionary
- No analysis for positional indexes, etc.

Pointers: two conflicting forces

- A term like Calpurnia occurs in maybe one doc out of a million - would like to store this pointer using $\log_2 1M \sim 20$ bits.
- A term like the occurs in virtually every doc, so 20 bits/pointer is too expensive.
  - Prefer 0/1 vector in this case.

Postings file entry

- Store list of docs containing a term in increasing order of doc id.
  - Brutus [33,47,154,159,202 ...]
  - Consequence: suffices to store gaps.
    - 33[14], 107, 5, 43 ...
  - Hope: most gaps encoded with far fewer than 20 bits.

Variable encoding

- For Calpurnia, will use ~20 bits/gap entry.
- For the, will use ~1 bit/gap entry.
- If the average gap for a term is $G$, want to use $-\log_2 G$ bits/gap entry.
- Key challenge: encode every integer (gap) with ~as few bits as needed for that integer.

$\gamma$ codes for gap encoding (Elias)

- Represent a gap $G$ as the pair $<\text{length}, \text{offset}>$
- length is in [unary] and uses $\lceil \log_2 G \rceil + 1$ bits to specify the length of the binary encoding of
- offset = $G - 2^{\lceil \log_2 G \rceil}$ in binary.

Recall that the unary encoding of $x$ is a sequence of $x$ 1’s followed by a 0.

$\gamma$ codes for gap encoding

- e.g., 9 represented as $<1110,001>$.
- 2 is represented as $<10,1>$.
- Exercise: does zero have a $\gamma$ code?
- Encoding $G$ takes $2 \lceil \log_2 G \rceil + 1$ bits.
  - $\gamma$ codes are always of odd length.
Exercise

- Given the following sequence of $\gamma$-coded gaps, reconstruct the postings sequence:
  
  $11100111010101111101101111011101110111011$

  From these $\gamma$-decode and reconstruct gaps, then full postings.

What we’ve just done

- Encoded each gap as tightly as possible, to within a factor of 2.
- For better tuning (and a simple analysis) - need a handle on the distribution of gap values.

Zipf’s law

- The $k$th most frequent term has frequency proportional to $1/k$.
- Use this for a crude analysis of the space used by our postings file pointers.
  - Not yet ready for analysis of dictionary space.

Zipf’s law log-log plot

Rough analysis based on Zipf

- The $i$th most frequent term has frequency proportional to $1/i$.
- Let this frequency be $c/i$.
- Then $\sum_{i=1}^{\infty} c/i = L$.
- The $k$th Harmonic number is $H_k = \sum_{i=1}^{k} 1/i$.
- Thus $c = 1/H_m$, which is $\sim 1/\ln m = 1/\ln(500k) \sim 1/13$.
- So the $i$th most frequent term has frequency roughly $1/13i$.

Postings analysis contd.

- Expected number of occurrences of the $i$th most frequent term in a doc of length $L$ is:
  
  $Lc/i \sim L/13i \sim 76/i$ for $L=1000$.

  Let $J = Lc \sim 76$.
  Then the $J$ most frequent terms are likely to occur in every document.
  Now imagine the term-document incidence matrix with rows sorted in decreasing order of term frequency:
Rows by decreasing frequency

- $J$ most frequent terms.
- $J$ next most frequent terms.
- $J$ next most frequent terms.
- etc.

$J$-row blocks

- In the $i$th of these $J$-row blocks, we have $J$ rows each with $n/i$ gaps of $i$ each.
- Encoding a gap of $i$ takes us $2\log_2 i + 1$ bits.
- So such a row uses space ~ $(2n \log_2 i)/i$ bits.
- For the entire block, $(2n \log_2 i)/i$ bits, which in our case is ~ $1.5 \times 10^8 (\log_2 i)/i$ bits.
- Sum this over $i$ from 1 up to $m/J = 500K/76$ ~ $6500$. (Since there are $m/J$ blocks.)

Exercise

- Work out the above sum and show it adds up to about $53 \times 150 \text{Mbits}$, which is about 1GB.
- So we’ve taken 6GB of text and produced from it a 1GB index that can handle Boolean queries!

Make sure you understand all the approximations in our probabilistic calculation.

Caveats

- This is not the entire space for our index:
  - does not account for dictionary storage – next up;
  - as we get further, we’ll store even more stuff in the index.
- Assumes Zipf’s law applies to occurrence of terms in docs.
- All gaps for a term taken to be the same.
- Does not talk about query processing.

More practical caveat

- $\gamma$ codes are neat but in reality, machines have word boundaries – 16, 32 bits etc.
  - Compressing and manipulating at individual bit-granularity is overkill in practice
  - Slows down architecture
  - In practice, simpler word-aligned compression (see Scholer reference) better

Word-aligned compression

- Simple example: fix a word-width (say 16 bits)
- Dedicate one bit to be a continuation bit $c$.
- If the gap fits within 15 bits, binary-encode it in the 15 available bits and set $c=0$.
- Else set $c=1$ and use additional words until you have enough bits for encoding the gap.
Exercise
How would you adapt the space analysis for $\gamma$-coded indexes to the scheme using continuation bits?

Exercise (harder)
How would you adapt the analysis for the case of positional indexes?

Intermediate step: forget compression. Adapt the analysis to estimate the number of positional postings entries.

Dictionary and postings files

Inverted index storage
Have estimated pointer storage
Next up: Dictionary storage
- Dictionary in main memory, postings on disk
  - This is common, especially for something like a search engine where high throughput is essential, but can also store most of it on disk with small, in-memory index
- Tradeoffs between compression and query processing speed
- Cascaded family of techniques

How big is the lexicon $V$?
- Grows (but more slowly) with corpus size
- Empirically okay model:
  \[ m = k N^b \]
  where $b \approx 0.5$, $k \approx 30–100$; $N = \#$ tokens
- For instance TREC disks 1 and 2 (2 Gb; 750,000 newswire articles): ~ 500,000 terms
- $V$ is decreased by case-folding, stemming
- Indexing all numbers could make it extremely large (so usually don’t)
- Spelling errors contribute a fair bit of size

Dictionary storage - first cut
- Array of fixed-width entries
  - 500,000 terms; 28 bytes/term = 14MB.

<table>
<thead>
<tr>
<th>Terms</th>
<th>Freq.</th>
<th>Postings ptr.</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td>999,712</td>
<td></td>
</tr>
<tr>
<td>aardvark</td>
<td>71</td>
<td></td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td></td>
</tr>
<tr>
<td>zzzz</td>
<td>99</td>
<td></td>
</tr>
</tbody>
</table>

(Allows for fast binary search into dictionary)

20 bytes
4 bytes each
Exercises

- Is binary search really a good idea?
- What are the alternatives?

Compressing the term list

- Store dictionary as a (long) string of characters:
  - Pointer to next word shows end of current word
  - Hope to save up to 60% of dictionary space.

<table>
<thead>
<tr>
<th>Term</th>
<th>Frequency</th>
<th>Postings ptr.</th>
<th>Term ptr.</th>
</tr>
</thead>
<tbody>
<tr>
<td>systyleszygetcysztygalysszyzbelytessczechomsy...</td>
<td>33</td>
<td>29</td>
<td>44</td>
</tr>
<tr>
<td>126</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- Binary search these pointers.

- Total string length = 500K x 8B = 4MB
- Pointers resolve 4M positions: \( \log_2 4M = 22 \text{ bits} = 3 \text{ bytes} \)

Fixed-width terms are wasteful

- Most of the bytes in the Term column are wasted – we allot 20 bytes for 1 letter terms.
  - And still can’t handle supercalifragilisticexpialidocious.
- Written English averages \( \frac{4}{3} \) characters.
  - Exercise: Why is/isn’t this the number to use for estimating the dictionary size?
  - Short words dominate token counts.
  - Average word in English: \( \frac{8}{3} \) characters.

Total space for compressed list

- 4 bytes per term for Frequency.
- 4 bytes per term for pointer to Postings.
- 3 bytes per term pointer
- Avg. 8 bytes per term in term string
- 500K terms \( \Rightarrow 9.5 \text{ MB} \)

Net

- Where we used 3 bytes/pointer without blocking
  - Example below: \( k=4 \).
  - Now avg. 11 bytes/term, not 20.

- Need to store term lengths (1 extra byte)
  - \( 3 \times 4 = 12 \) bytes for \( k=4 \) pointers,
  - now we use \( 3+4=7 \) bytes for 4 pointers.

- Shaved another \(~0.5\)MB; can save more with larger \( k \).
- Why not go with larger \( k \)?
Exercise

- Estimate the space usage (and savings compared to 9.5MB) with blocking, for block sizes of $k = 4$, 8 and 16.

Impact on search

- Binary search down to 4-term block;
- Then linear search through terms in block.
- 8 documents; binary tree ave. = 2.6 compares
- Blocks of 4 (binary tree), ave. = 3 compares

\[
= (1+2+4+3+4)B = (1+2+2+3+2+4+5)B
\]

Exercise

- Estimate the impact on search performance (and slowdown compared to $k=1$) with blocking, for block sizes of $k = 4$, 8 and 16.

Total space

- By increasing $k$, we could cut the pointer space in the dictionary, at the expense of search time; space 9.5MB $\rightarrow$ ~8MB
- Net – postings take up most of the space
  - Generally kept on disk
  - Dictionary compressed in memory

Some complicating factors

- Accented characters
  - Do we want to support accent-sensitive as well as accent-insensitive characters?
  - E.g., query *resume* expands to *resume* as well as *résumé*
  - But the query *résumé* should be executed as only *résumé*
  - Alternative, search application specifies
- If we store the accented as well as plain terms in the dictionary string, how can we support both query versions?

Index size

- Stemming/case folding cut
  - number of terms by ~40%
  - number of pointers by 10-20%
  - total space by ~30%
- Stop words
  - Rule of 30: ~30 words account for ~30% of all term occurrences in written text
  - Eliminating 150 commonest terms from indexing will cut almost 25% of space
Extreme compression (see MG)

- Front-coding:
  - Sorted words commonly have long common prefix
    - store differences only
  - (for last \(k-1\) in a block of \(k\))

\[
8 \text{automata} \rightarrow \{8 \text{(automat)}\} 0 \text{e26ic3lion}
\]

Encodes \text{automat} Extra length beyond \text{automat}.

Begin to resemble general string compression.

Extreme compression

- Using (perfect) hashing to store terms “within” their pointers
  - not great for vocabularies that change.
- Large dictionary: partition into pages
  - use B-tree on first terms of pages
  - pay a disk seek to grab each page
  - if we’re paying 1 disk seek anyway to get the postings, “only” another seek/query term.

Compression: Two alternatives

- Lossless compression: all information is preserved, but we try to encode it compactly
  - What IR people mostly do
- Lossy compression: discard some information
  - Using a stopword list can be viewed this way
  - Techniques such as Latent Semantic Indexing (later) can be viewed as lossy compression
  - One could prune from postings entries unlikely to turn up in the top \(k\) list for query on word
    - Especially applicable to web search with huge numbers of documents but short queries (e.g., Carmel et al. SIGIR 2002)

Top \(k\) lists

- Don’t store all postings entries for each term
  - Only the “best ones”
  - Which ones are the best ones?
- More on this subject later, when we get into ranking

Resources

- MG 3.3, 3.4.