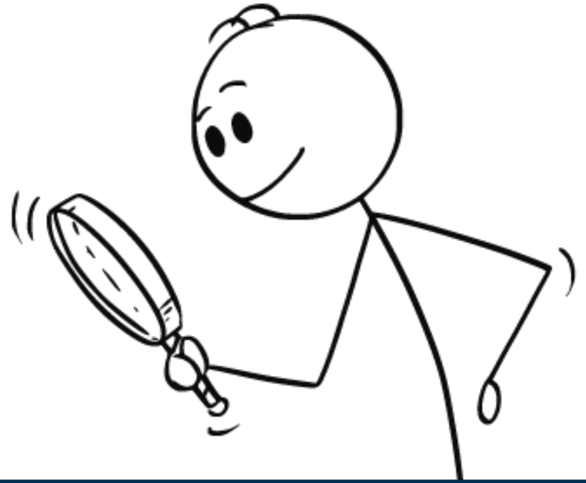


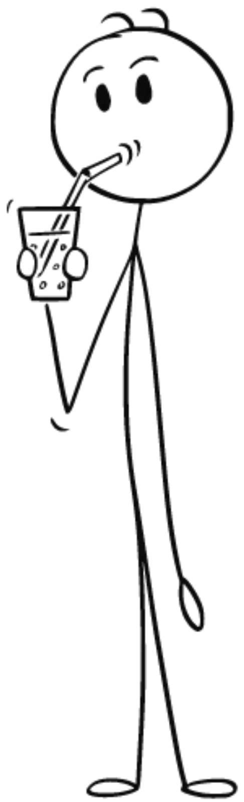
CSE 4/535 Information Retrieval

Sayantan Pal
PhD Student, Department of CSE
338Z Davis Hall

University at Buffalo
Department of CSE

Before we start

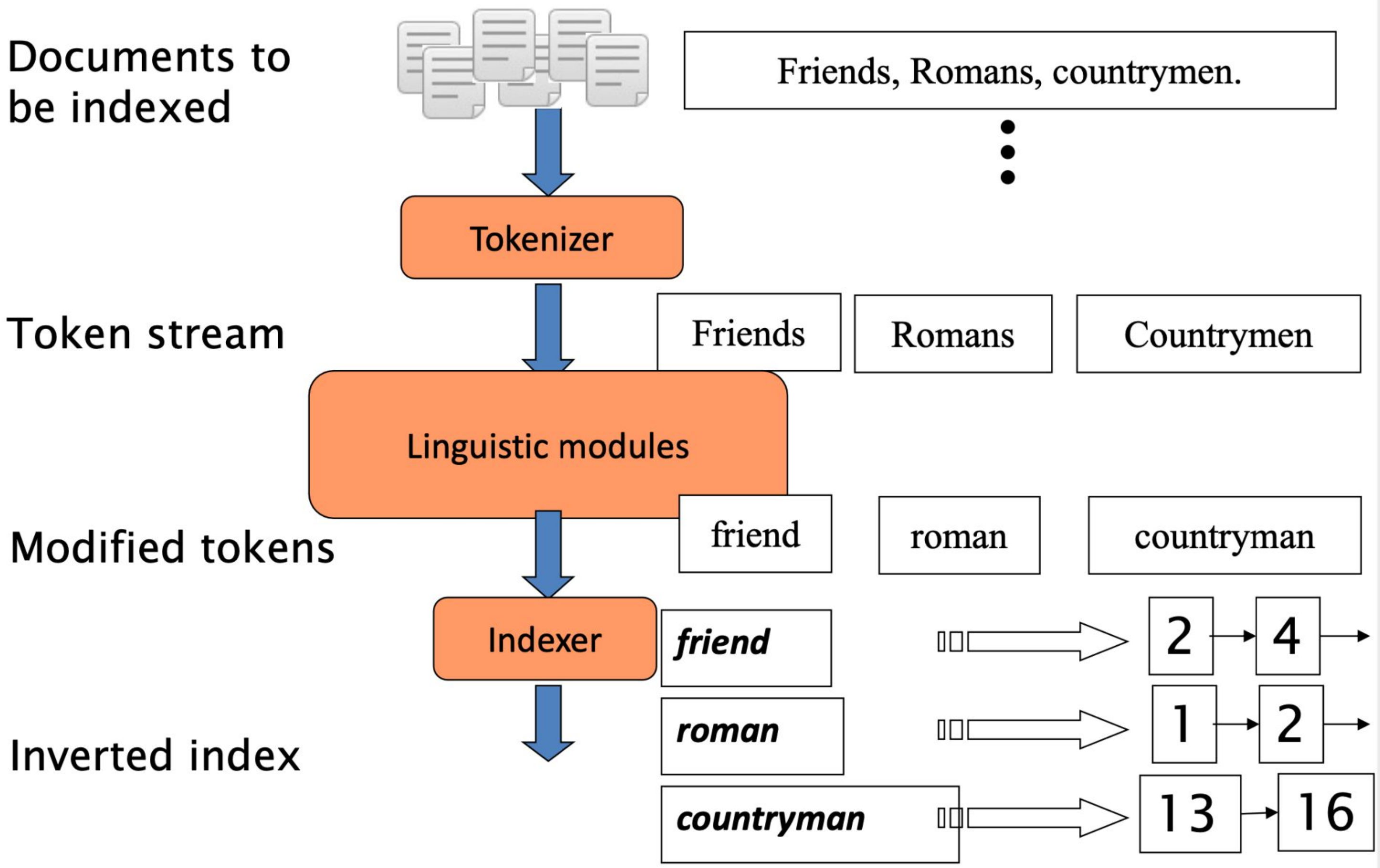
1. Project 1 released, due 27th September.
2. Join office hours if you have questions
3. Today's lecture is important for the first Midterm
    a. Stay focused!



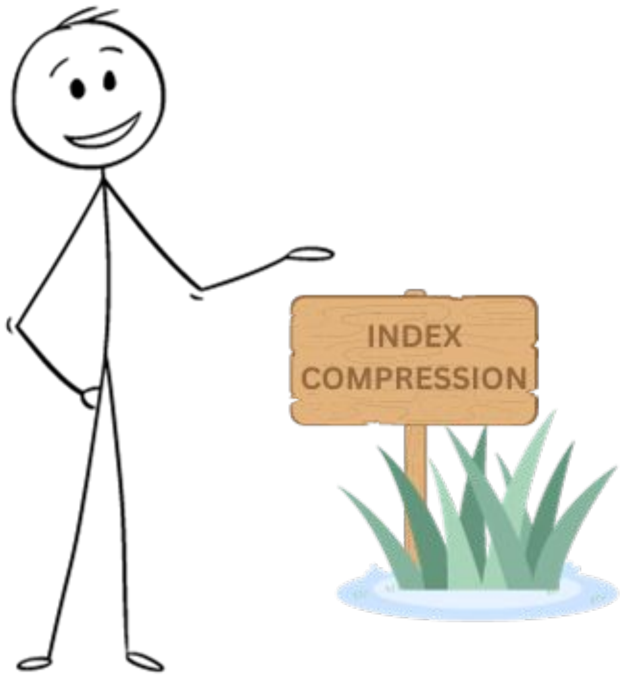
Recap - Previous Class

1. How to construct Index efficiently?
2. Disk vs memory.
3. RCV1 corpus
4. Algorithms for Indexing
5. Dynamic Indexing

Recall the basic indexing pipeline



Index Compression

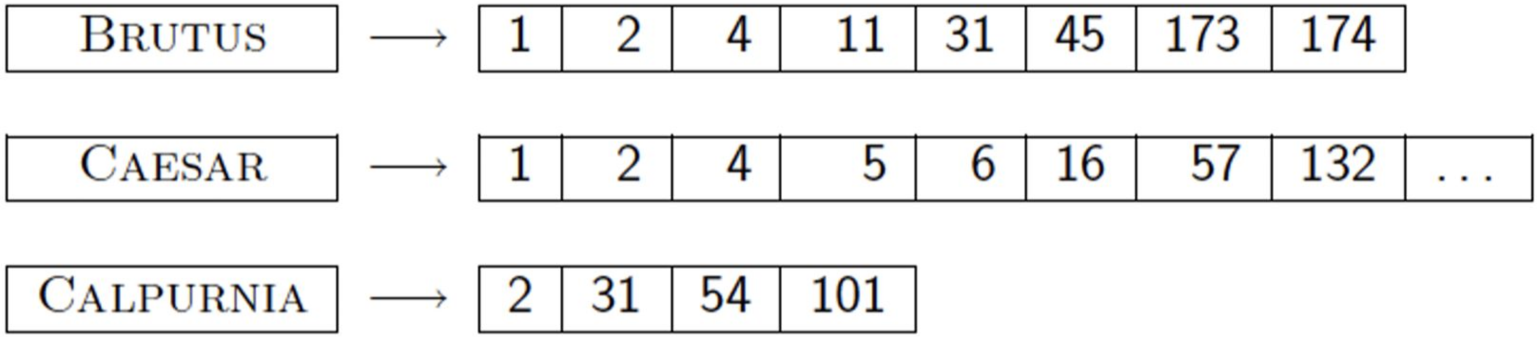


Index Compression….. Why?

- Why do we need compression?

Index Compression..... Why?

- Collection statistics in more detail (with RCV1)
  - How big will the dictionary and postings be?
- Dictionary compression
- Postings compression



Index Compression..... Why?

- Use less disk space
  - Save a little money
  - give users more space
- Keep more stuff in memory
- Increases speed
- Increase speed of data transfer from disk to memory
  - **read compressed data**
  - **decompress is faster** than reading uncompressed data
- Premise: Decompression algorithms are fast

Why compression for inverted indexes?

- Dictionary
  - Make it small enough to keep in main memory
  - Make it so small that you can keep some postings lists in main memory too
- Postings file(s)
  - Reduce disk space needed
  - Decrease time needed to read postings lists from disk
  - Large search engines keep a significant part of the postings in memory
    - Compression lets you keep more in memory

Recall Reuters RCV1

symbol	statistic	value
N	documents	800,000
L	avg. # tokens per document	200
M	term types	400,000
	avg. # bytes per token (incl. spaces/punct.)	6
	avg. # bytes per token (without spaces/punct.)	4.5
	avg. # bytes per term type	7.5
	non-positional postings	100,000,000

Index parameters vs. what we index

size of	word types (terms)			non-positional postings			positional postings		
	dictionary			non-positional index			positional index		
	Size (K)	∆%	cumul %	Size (K)	∆ %	cumul %	Size (K)	∆ %	cumul %
Unfiltered	484			109,971			197,879		
No numbers	474	-2	-2	100,680	-8	-8	179,158	-9	-9
Case folding	392	-17	-19	96,969	-3	-12	179,158	0	-9
30 stopwords	391	-0	-19	83,390	-14	-24	121,858	-31	-38
150 stopwords	391	-0	-19	67,002	-30	-39	94,517	-47	-52
stemming	322	-17	-33	63,812	-4	-42	94,517	0	-52

Vocabulary size vs. collection size

- How big is the term vocabulary?
  - That is, how many distinct words are there?

Vocabulary size vs. collection size

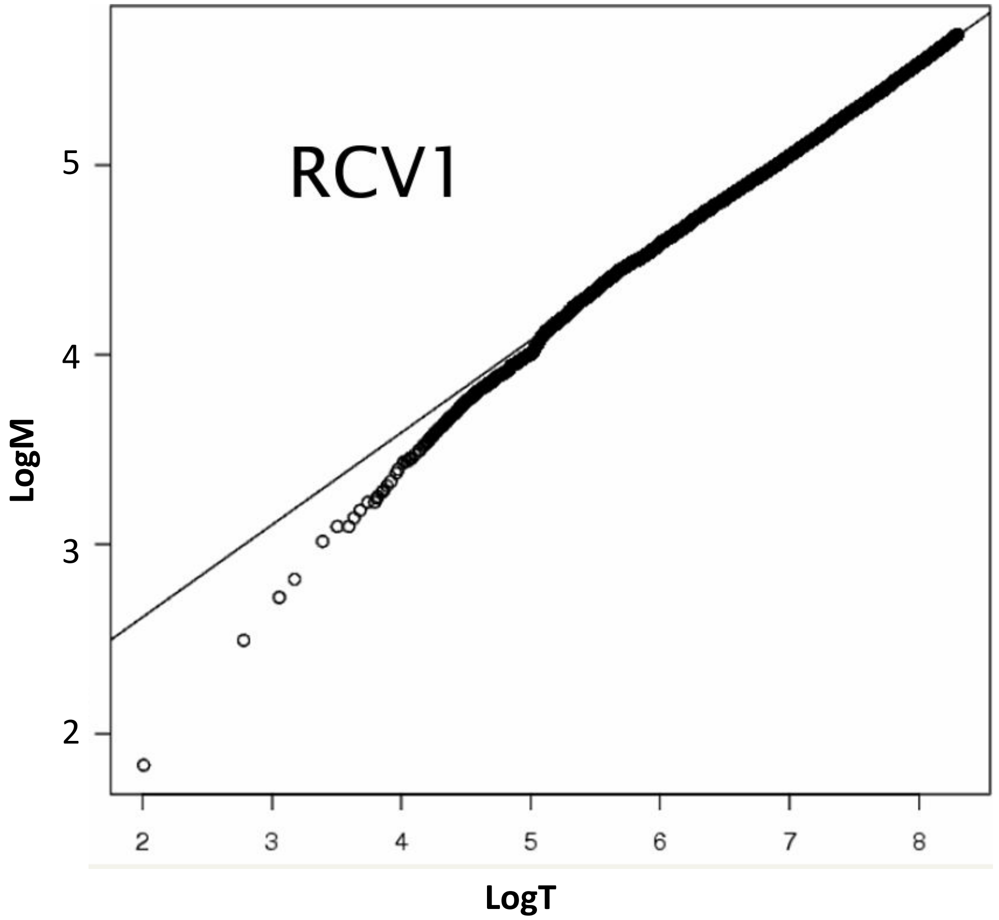
- How big is the term vocabulary?
  - That is, how many distinct words are there?
- Can we assume an upper bound?
  - Not really: At least $70^{20} = 10^{37}$ different words of length 20
- In practice, the vocabulary will keep growing with the collection size
  - Especially with Unicodes

Vocabulary size vs. collection size

- Heaps' law: $M = kT^b$
- M is the size of the vocabulary, T is the number of tokens in the collection
- Typical values: $30 \le k \le 100$ and $b \approx 0.5$
- In a log-log plot of vocabulary size M vs. T, Heaps' law predicts a line with slope about ½
- It is the simplest possible (linear) relationship between the two in log-log space
- $\log M = \log k + b \log T$
- An empirical finding ("empirical law")

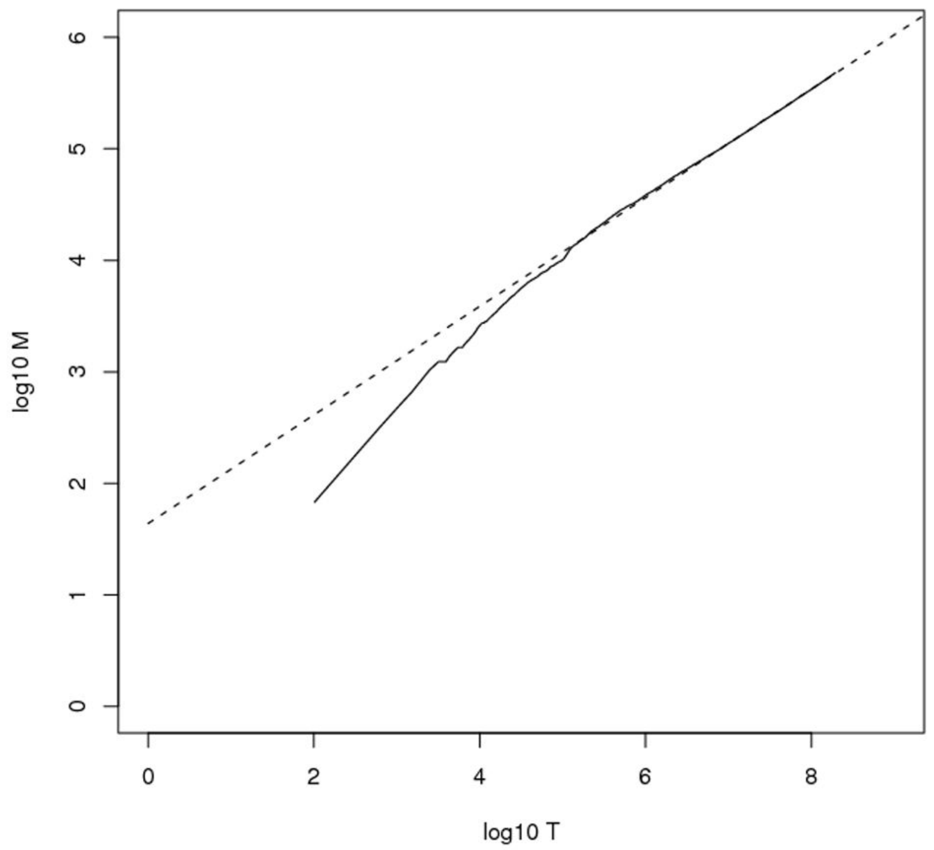
The number of dictionary terms

- Can’t use the size of a language vocabulary like the Oxford English Dictionary
- Many names of people, products, addresses
- Heaps’ Law Measures vocabulary size as function of collection size $M = kT^b$ Relationship Linear in log-log space
  - T = Number of tokens where $30 \leq k \leq 100$ (determined by collection, tokenization) b»0.5 For 1M tokens, Heaps predicts 38,323 terms, actual is 38,365



Heaps' Law

- For RCV1, the dashed line $\log_{10}M = 0.49 \log_{10}T + 1.64$ is the best least squares fit.
- Thus, $M = 10^{1.64}T^{0.49}$ so **$k = 10^{1.64} \approx 44$** and **$b = 0.49$**. Good empirical fit for Reuters RCV1!
- For first 1,000,020 tokens, law predicts 38,323 terms; actually, 38,365 terms



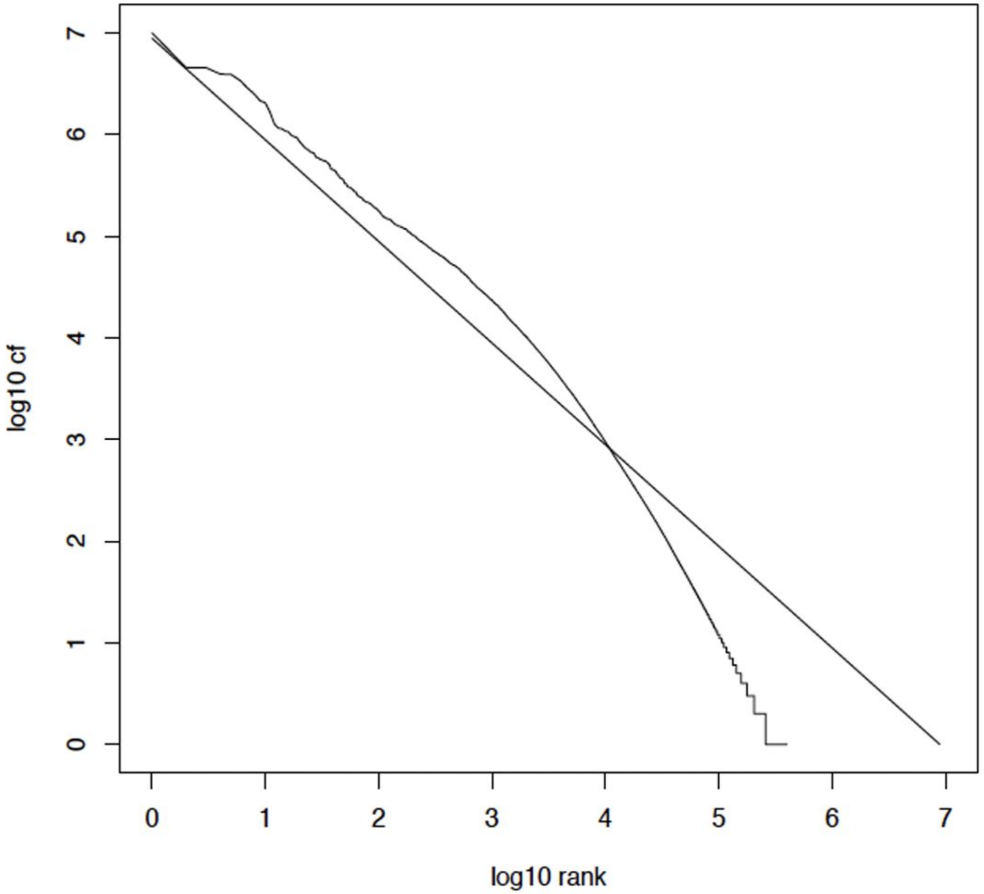
Zipf's law

1. Heaps' law gives the vocabulary size in collections.
2. We also study the **relative frequencies of terms** (distribution of terms in a document).
3. In natural language, there are a few very frequent terms and very many very rare terms.
4. Zipf's law: The i^{th} most frequent term has frequency proportional to 1/i.
5. $cf_i \propto 1/i = K/i$ where K is a normalizing constant
6. cf_i is collection frequency: the number of occurrences of the term t_i in the collection.

Zipf consequences

1. If the most frequent term (the) occurs cf_1 times
   a. Then the second most frequent term (of) occurs $cf_1/2$ times
   b. The third most frequent term (and) occurs $cf_1/3$ times …
2. Equivalent: $cf_i = K/i$ where K is a normalizing factor, so
   a. $\log cf_i = \log K - \log i$
   b. Linear relationship between $\log cf_i$ and $\log i$
3. Another power law relationship

Zipf's law for Reuters RCV1



the frequency of any word is **inversely proportional** to its rank in the frequency table.

Compression

1. Now, we will consider compressing the space for the dictionary and postings. We'll do:
    a. Basic Boolean index only
    b. No study of positional indexes, etc.
2. But these ideas can be extended

Dictionary Compression

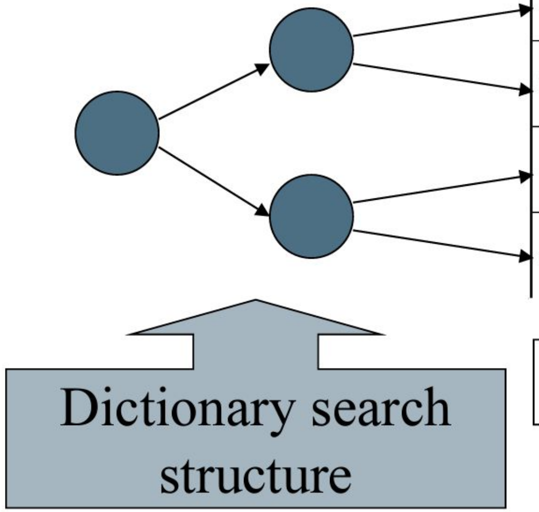
Why compress the dictionary?

1. Search begins with the dictionary
2. We want to keep it in memory
3. Memory footprint competition with other applications
4. Embedded/mobile devices may have very little memory
5. Even if the dictionary isn’t in memory, we want it to be small for a fast search startup time

Dictionary storage –naïve version

- Array of fixed-width entries
  - ~400,000 terms (RCV corpus) ; 28 bytes/term = 11.2 MB.

Terms	Freq.	Postings ptr.
a	656,265	
aachen	65	
….	….	
zulu	221	



20 bytes

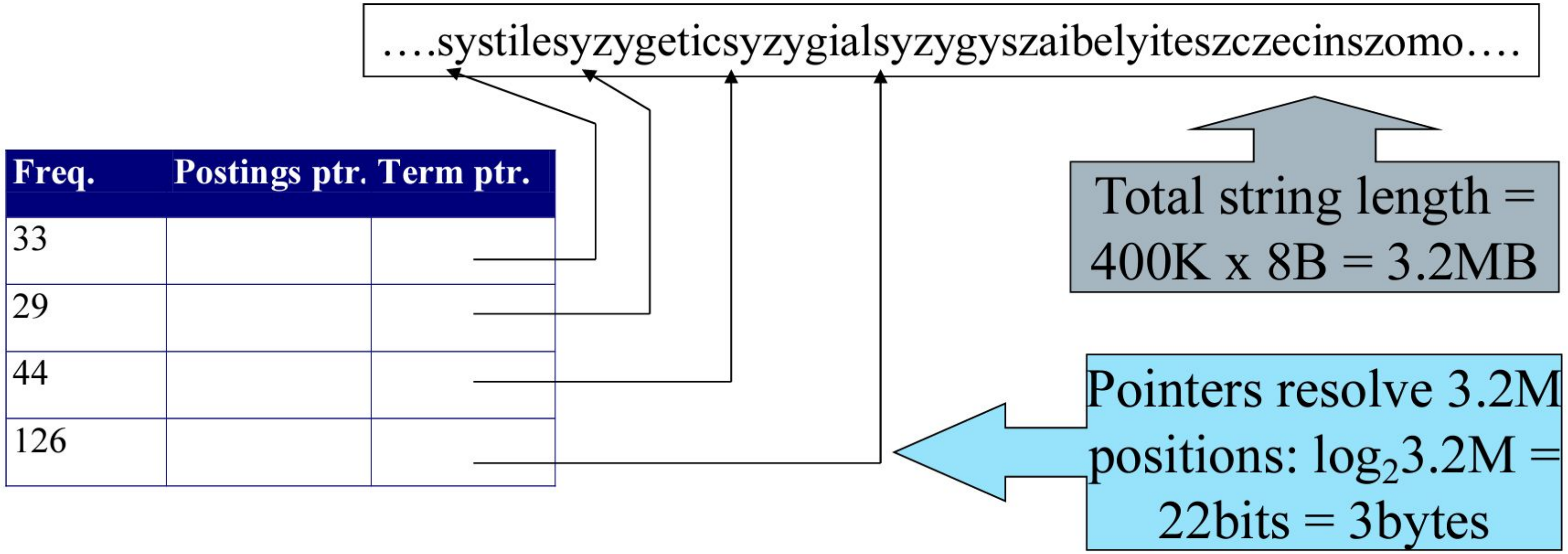
4 bytes each

Fixed-width terms are wasteful

- Most of the bytes in the Term column are wasted we allot 20 bytes for 1 letter terms.
    - And we still can’t handle floccinaucinihilipilification or hydrochlorofluorocarbons.
- Written English averages ~4.5 characters/word.
- Ave. dictionary word in English: ~8 characters
    - How do we use ~8 characters per dictionary term?
- Short words dominate token counts but not type average.

Compressing the term list: Dictionary-as-a-String

- 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



Freq.	Postings ptr.	Term ptr.
33		
29		
44		
126		

Space for dictionary as a string

1. 4 bytes per term for Freq.
2. 4 bytes per term for pointer to Postings.
3. 3 bytes per term pointer
4. Avg. 8 bytes per term in term string
5. 400K terms x 19 =>7.6 MB (against 11.2MB for fixed width)
6. Now avg. 11 bytes/term, not 20

Blocking

- Store pointers to every kth term string.
    - Example below: k=4.
- Need to store term lengths (1 extra byte)

….7*systile*9*syzygetic*8*syzygial*6*syzygy*11*szaibelyite*8*szczecin*9*szomo*….

Freq.	Postings ptr.	Term ptr.
33		
29		
44		
126		
7		

Save 9 bytes on 3 pointers.

Lose 4 bytes on term lengths.

Blocking Net Gains

- Example for block size k= 4
- Where we used 3 bytes/pointer without blocking
  - 3 x 4 = 12 bytes,
- now we use 3 + 4 = 7 bytes.

Shaved another ~0.5MB. This reduces the size of the dictionary from 7.6 MB to 7.1 MB. We can save more with larger k.

Blocking Net Gains

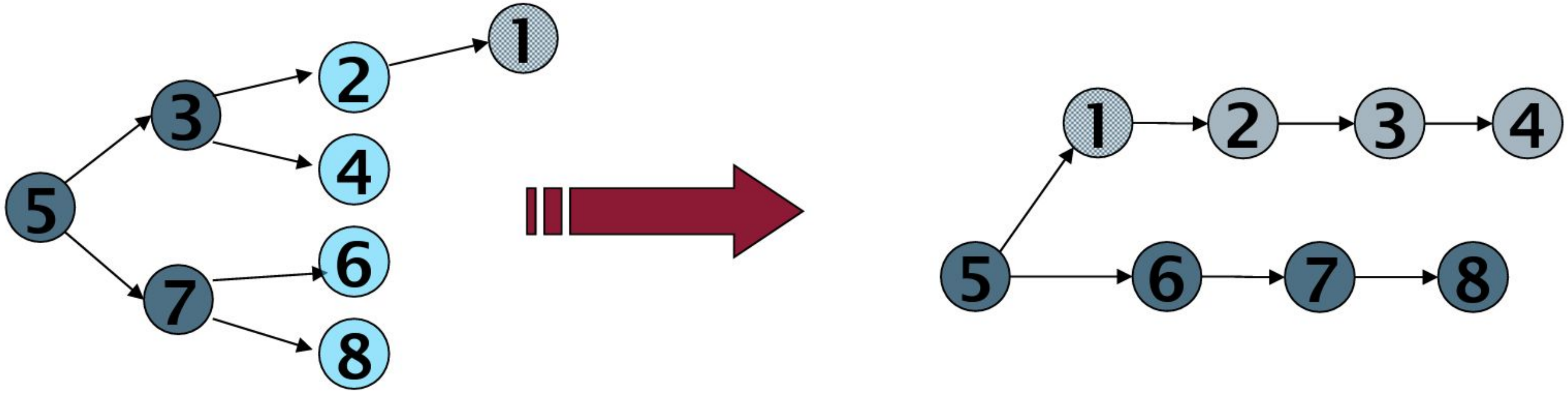
- Example for block size k= 4
- Where we used 3 bytes/pointer without blocking
    - 3 x 4 = 12 bytes,
- now we use 3 + 4 = 7 bytes.

Shaved another ~0.5MB. This reduces the size of the dictionary from 7.6 MB to 7.1 MB. We can save more with larger k.

Why not larger k?

Impact on search

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



$= (1+2\cdot2+4\cdot3+4)/8$ $=(1+2\cdot2+2\cdot3+2\cdot4+5)/8$

Numerator: Total number of comparisons to reach each node at every level; there are 4 levels
Denominator: total number of nodes (or documents)

Front coding

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

8***automata***8***automate***9***automatic***10***automation***

→8***automat**** ***a***1◊***e***2◊***ic***3◊***ion***

Encodes prefix ***automat***

Extra length beyond ***automat.***

Begins to resemble general string compression

RCV1 dictionary compression summary

Technique	Size in MB
Fixed width	11.2
Dictionary-as-String with pointers to every term	7.6
+ blocking, $k = 4$	7.1
+ blocking + front coding	5.9

POSTINGS COMPRESSION

Postings compression

- The postings file is much larger than the dictionary, factor of at least 10, often over 100 times larger
- Key: store each posting compactly.
- A posting for our purposes is a docID.
- For Reuters (800,000 documents), we would use 32 bits per docID when using 4-byte integers.
- Alternatively, we can use $\log_2 800{,}000 \approx 20$ bits per docID.
- Our goal: use far fewer than 20 bits per docID.

Postings: two conflicting forces

1. A term like **arachnocentric** occurs in maybe one doc out of a million –we would like to store this posting using $\log_2 1M \approx 20$ bits.
2. A term like **the** occurs in virtually every doc, so 20 bits/posting ≈ 2MB is too expensive.

Gap encoding of postings file entries

- We store the list of docs containing a term in increasing order of docID.
  - ***computer***: 33,47,154,159,202 …
- <u>Consequence</u>: it suffices to store *gaps*.
  - 33,14,107,5,43 …
- <u>Hope</u>: most gaps can be encoded/stored with far fewer than 20 bits.
  - Especially for common words

Three postings entries

	encoding	postings list									
THE	docIDs	...		283042		283043		283044		283045	...
	gaps				1		1		1		...
COMPUTER	docIDs	...		283047		283154		283159		283202	...
	gaps				107		5		43		...
ARACHNOCENTRIC	docIDs	252000		500100							
	gaps	252000	248100								

Variable length encoding

- Aim:
  - For **arachnocentric**, we will use ~20 bits/gap entry.
  - For **the**, we will use ~1 bit/gap entry.
- If the average gap for a term is G, we want to use ~$\log_2 G$ bits/gap entry.
- Key challenge: encode every integer (gap) with about as few bits as needed for that integer.
- This requires a variable length encoding
- Variable length codes achieve this by using short codes for small numbers

Unary code

- Represent n as n 1s with a final 0.
- Unary code for 3 is 1110.
- Unary code for 40 is 11111111111111111111111111111111111111110 .
- Unary code for 80 is:
  111111111111111111111111111111111111111111111111111111111111111111111111111111110
- This doesn’t look promising

ɣ(gamma) codes for gap encoding

Length	Offset

- Represent a gap Gas the pair <length,offset>
- Offset: binary encoding of G, drop the leading 1.
- Length:is in unary and uses floor($\log_2$G)+1 bits to specify the length of the binary encoding of offset

ɣ(Gamma) codes

- We can compress better with bit-level codes
  - The Gamma code is the best known of these.
- Represent a gap Gas a pair length and offset
- offset is G in binary, with the leading bit cut off
  - For example 13 → 1101 → 101 (13 in binary is 1101)
- length is the length of offset
  - For 13 (offset 101), this is 3.
- We encode length with unary code: 1110.
- Gamma code of 13 is the concatenation of length and offset: 1110101

Gamma code examples

number	length	offset	γ-code
0			none
1	0		0
2	10	0	10,0
3	10	1	10,1
4	110	00	110,00
9	1110	001	1110,001
13	1110	101	1110,101
24	11110	1000	11110,1000
511	111111110	11111111	111111110,11111111
1025	11111111110	0000000001	11111111110,0000000001

Exercise

- Decoding:
  - read unary code up to 0 that terminates it
  - Tells you how long the offset is
- Given the following sequence of γ–coded gaps, reconstruct the postings sequence:

1110001110101011111101101111011

From these γ–decode and reconstruct gaps, then full postings.

9, 6, 3, etc.

Variable Byte (VB) codes

- For a gap value G, we want to use close to the fewest bytes needed to hold $\log_2 G$ bits
- Begin with one byte to store G and dedicate 1 bit in it to be a continuation bit c
- If G≤127, binary-encode it in the 7 available bits and set c =1
- Else encode G’s lower-order 7 bits and then use additional bytes to encode the higher order bits using the same algorithm
- At the end set the continuation bit of the last byte to 1 (c=1) and for the other bytes c = 0.

Example

docIDs	824	829	215406
gaps		5	214577
VB code	00000110 10111000	10000101	00001101 00001100 10110001

Postings stored as the byte concatenation
000001101011100010000101000011010000110010110001

Key property: VB-encoded postings are uniquely prefix-decodable.

824 is binary 1100111000

For a small gap (5), VB uses a whole byte.

RCV1 compression

Data structure	Size in MB
dictionary, fixed-width	11.2
dictionary, term pointers into string	7.6
with blocking, k = 4	7.1
with blocking & front coding	5.9
collection (text, xml markup etc)	3,600.0
collection (text)	960.0
Term-doc incidence matrix	40,000.0
postings, uncompressed (32-bit words)	400.0
postings, uncompressed (20 bits)	250.0
postings, variable byte encoded	116.0
postings, γ–encoded	101.0

References

1. Slides provided by Sougata Saha (Instructor, Fall 2022 - CSE 4/535)
2. Materials provided by Dr. Rohini K Srihari
3. https://nlp.stanford.edu/IR-book/information-retrieval-book.html