CS3245

Information Retrieval

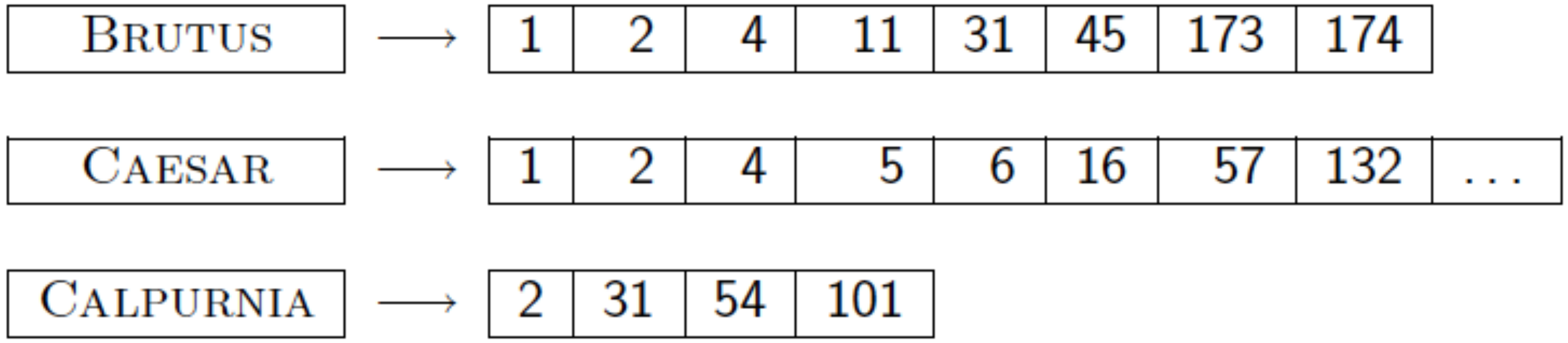
6

Lecture 6: Index Compression

Last Time: index construction

- Sort-based indexing
  - Blocked Sort-Based Indexing
    - Merge sort is effective for disk-based sorting (avoid seeks!)
  - Single-Pass In-Memory Indexing
    - No global dictionary - Generate separate dictionary for each block
    - Don’t sort postings - Accumulate postings as they occur
- Distributed indexing using MapReduce
- Dynamic indexing: Multiple indices, logarithmic merge

Today: Cmprssn


BRUTUS → 1 2 4 11 31 45 173 174
CAESAR → 1 2 4 5 6 16 57 132 . . .
CALPURNIA → 2 31 54 101

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

Vocabulary 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 relationship between the two in log-log space
  - An empirical finding ("empirical law")

Heaps' Law

For RCV1, the dashed line

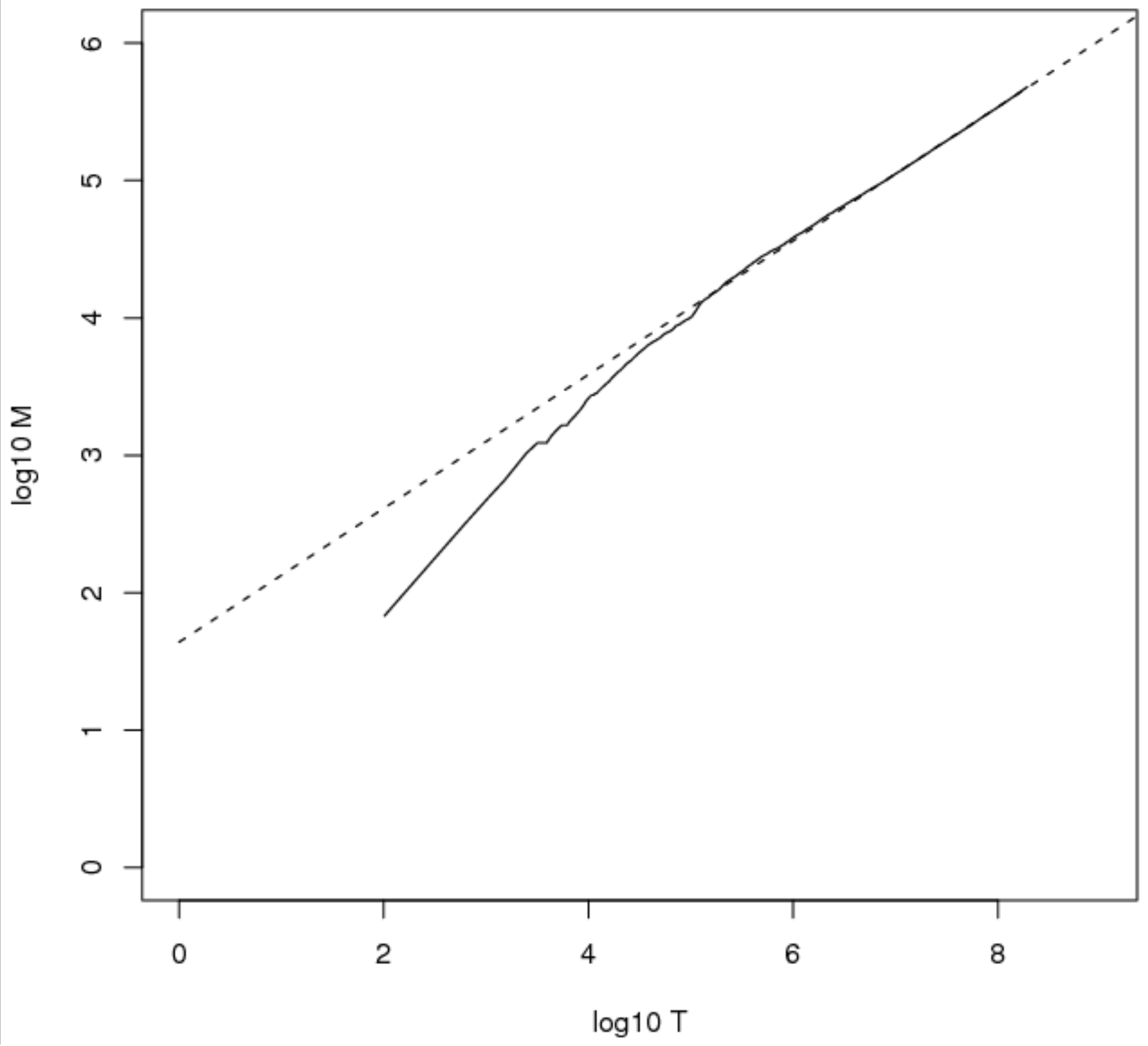
$\log_{10} M = 0.49 \log_{10} T + 1.64$

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



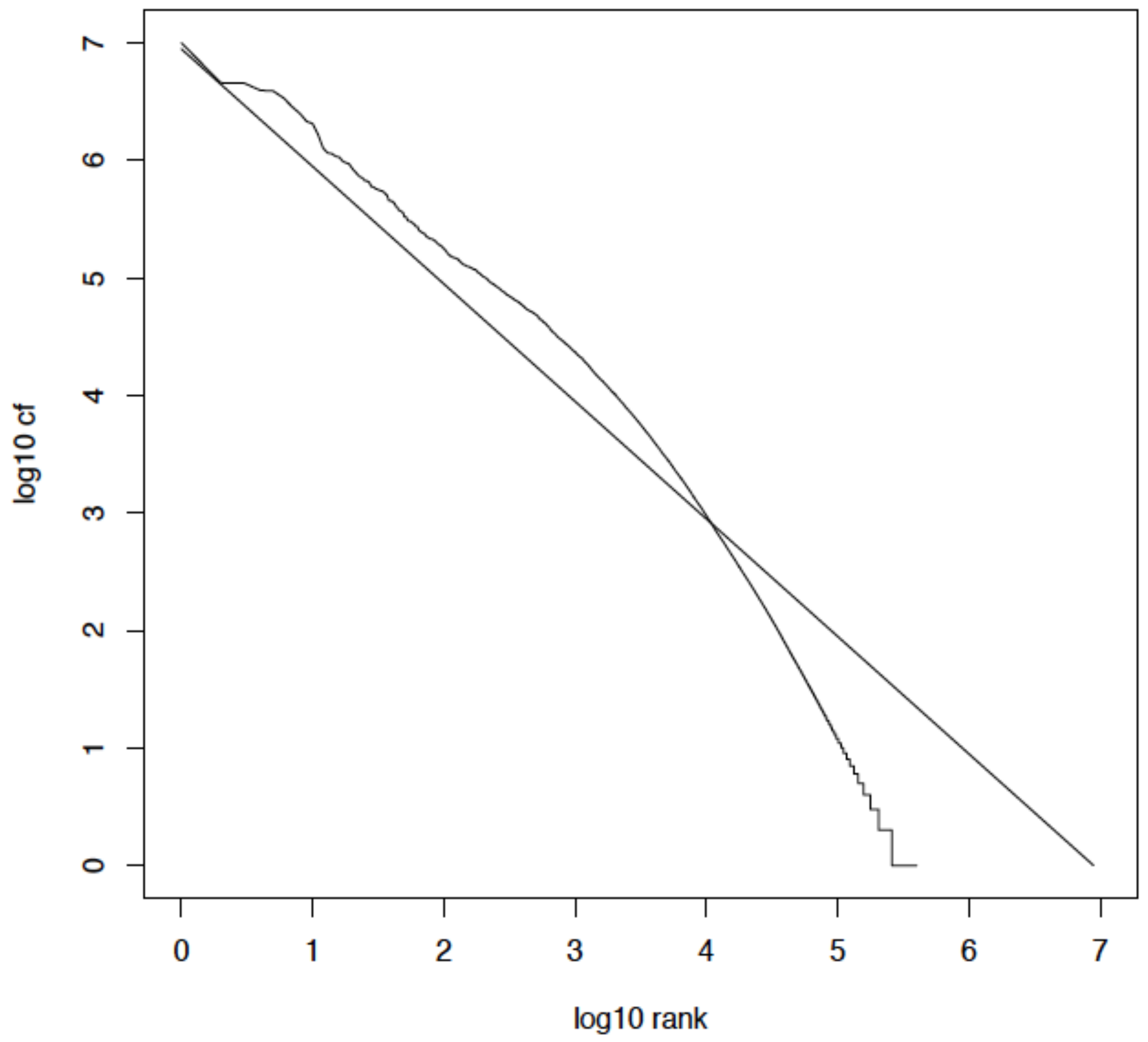
Zipf's law

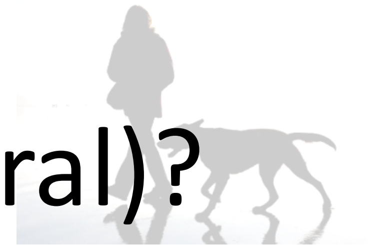
- How about the relative frequencies of terms?
- In natural language, there are a few very frequent terms and very many very rare terms.
- Zipf's law: The *i*th most frequent term has frequency proportional to 1/*i* .
- $\mathrm{cf}_i \propto 1/i = K/i$ where *K* is a normalizing constant
- cf_i is collection frequency (not document frequency): the number of occurrences of the term t_i in the collection.

Zipf consequences

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

Zipf' s law for Reuters RCV1





Why compression (in general)?

- Use less disk space
  - Saves a little money
- Keep more data in memory
  - Increases speed
- Increase speed of data transfer from disk to memory
  - [read compressed data | decompress] is faster than [read uncompressed data]
  - Premise: Decompression algorithms are fast
    - True of the decompression algorithms we use

Lossless vs. lossy compression

- Lossless compression: All information is preserved
  - What we mostly do in IR.
- Lossy compression: Discard some information
- Several of the preprocessing steps can be viewed as lossy compression: case folding, stop words, stemming, number elimination
- Later: Prune postings entries that are unlikely to turn up in the top *k* list for any query
  - Almost no loss quality for top *k* list

DICTIONARY COMPRESSION

Why compress the dictionary?

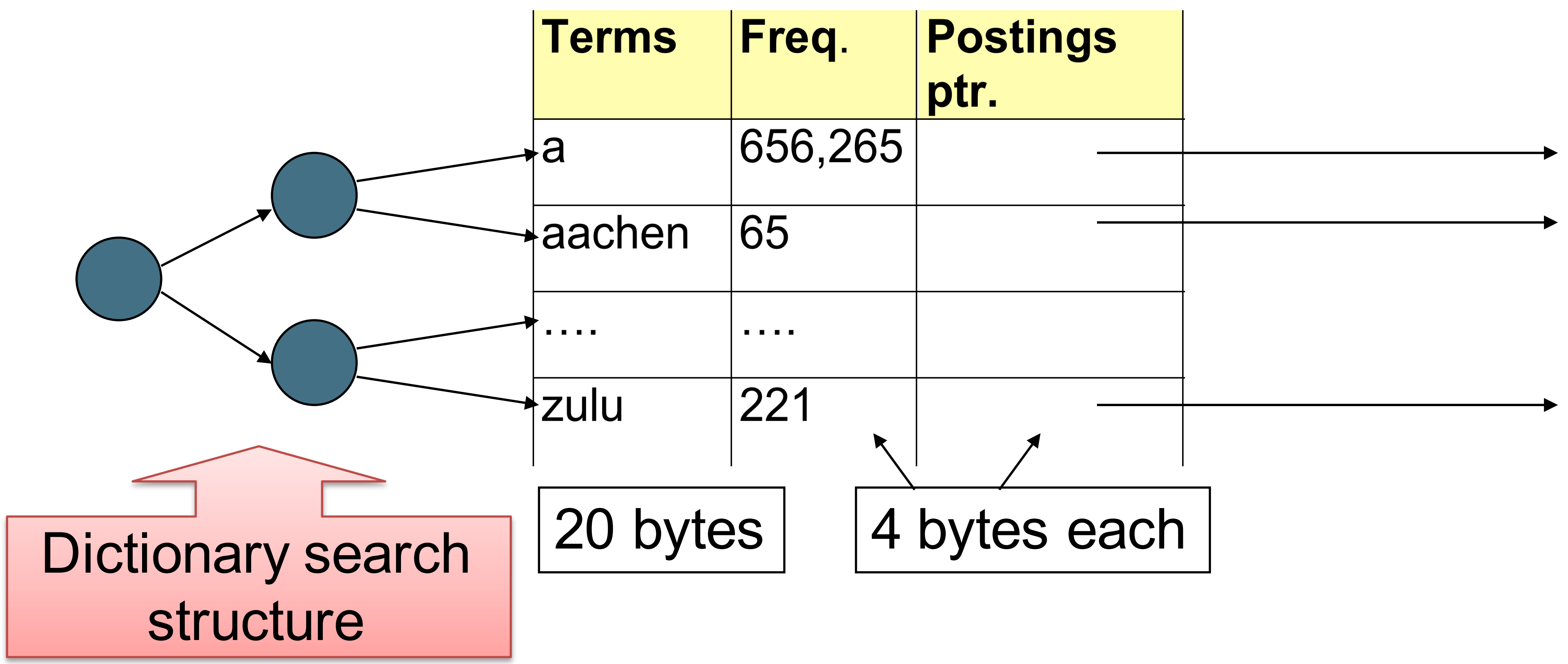
- Search begins with the dictionary
- We want to keep it in memory
- Memory footprint competition with other applications
- Embedded/mobile devices may have very little memory
- Even if the dictionary isn't in memory, we want it to be small for a fast search startup time

Compressing the dictionary is important

Dictionary storage - first cut

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

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

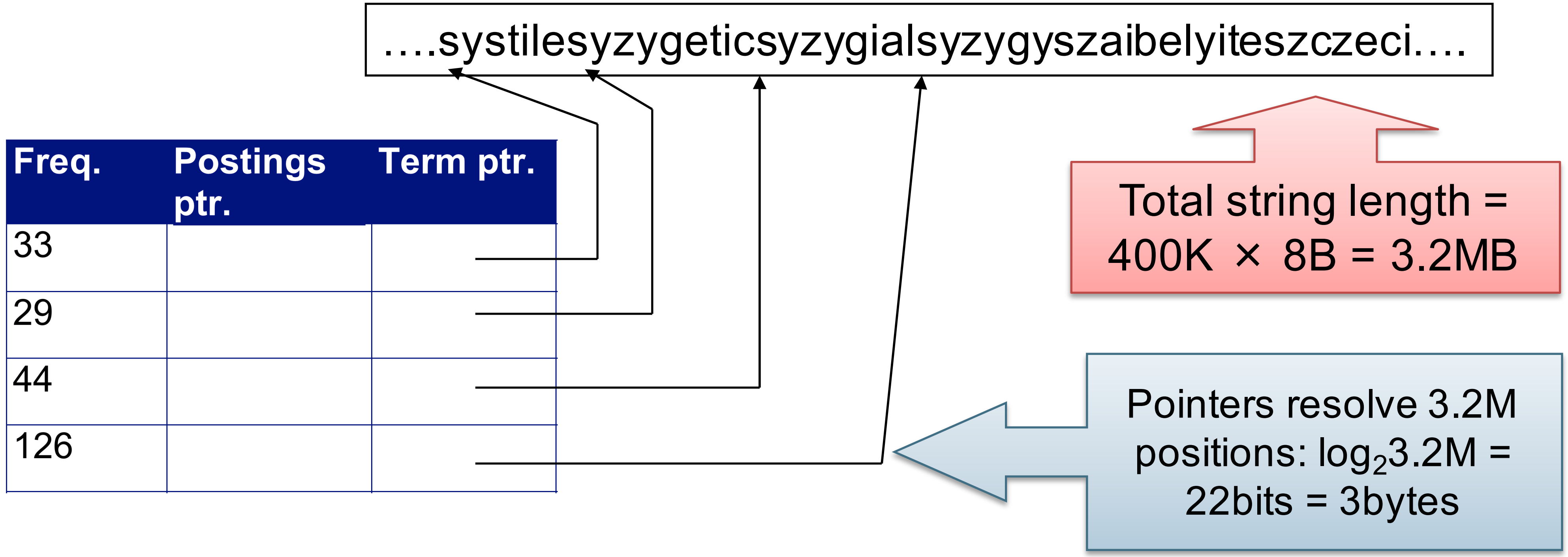


Fixed-width terms are wasteful

- Most of the bytes in the **Term** column are wasted – we allot 20 bytes for 1 letter terms.
  - And we still can't handle *supercalifragilisticexpialidocious* or *hydrochlorofluorocarbons.*
- Written English averages ~4.5 characters/word.
- Average 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.



….systilesyzygeticsyzygialsyzygyszaibelyiteszczeci….

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

Total string length = 400K × 8B = 3.2MB

Pointers resolve 3.2M positions: $\log_2 3.2M$ = 22bits = 3bytes

Space for dictionary as a string

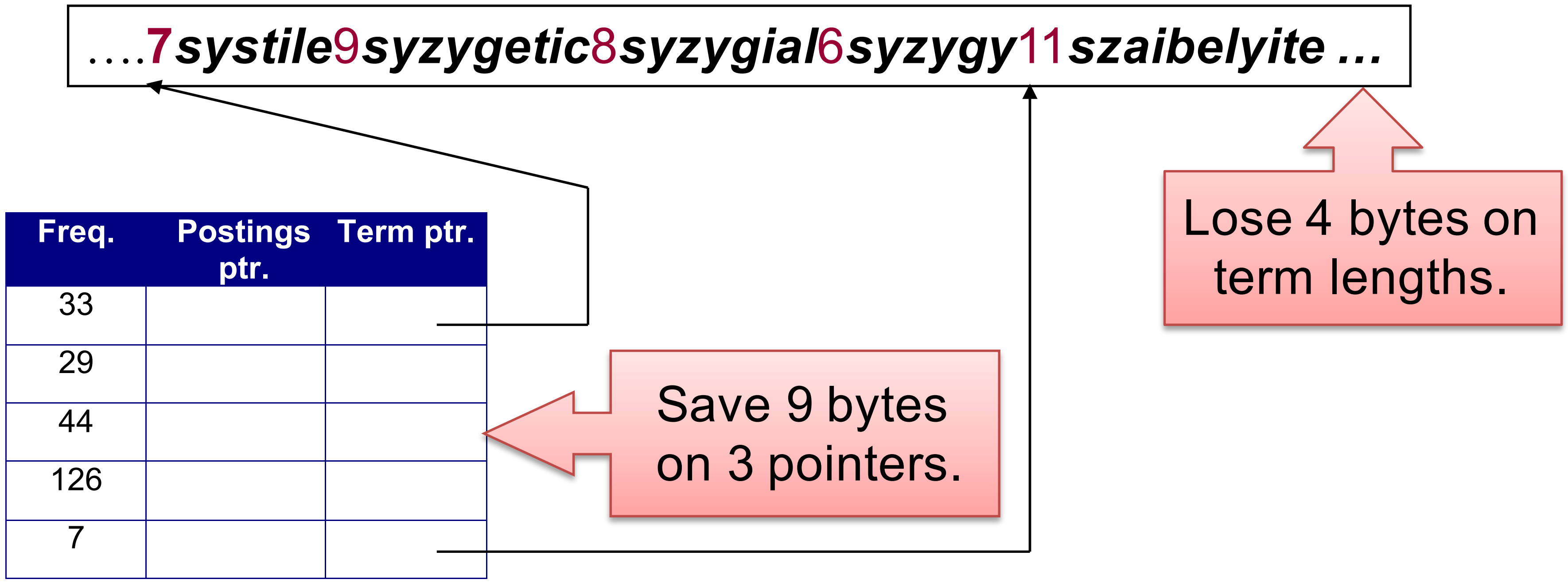
- 4 bytes per term for frequency
- 4 bytes per term for pointer to postings
- 3 bytes per term pointer

} Now avg. 11 bytes/term, not 20.

- Avg. 8 bytes per term in term string
- 400K terms × 19 ⇒ 7.6 MB (against 11.2MB for fixed width)

Blocking

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



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

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

Net Result

- Example for block size *k* = 4
- Where we used 3 bytes/pointer without blocking
  - 3 x 4 = 12 bytes,

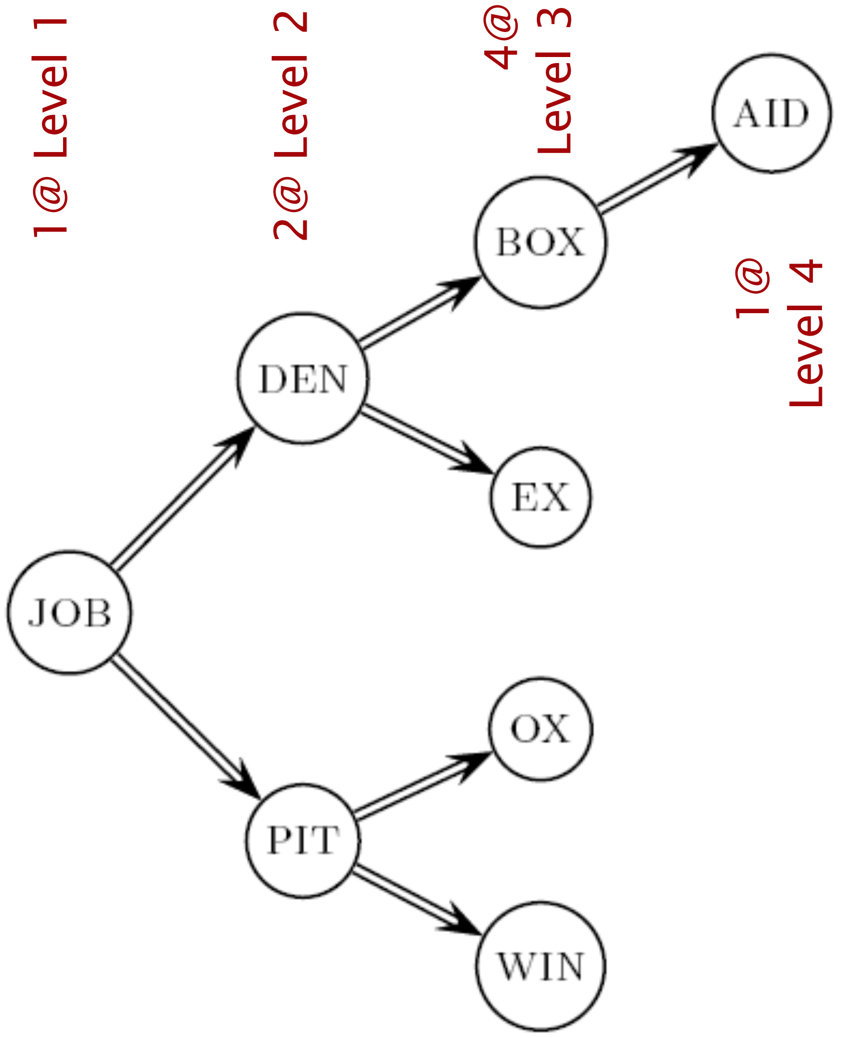
now we use 3 + 4 = 7 bytes.

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*.

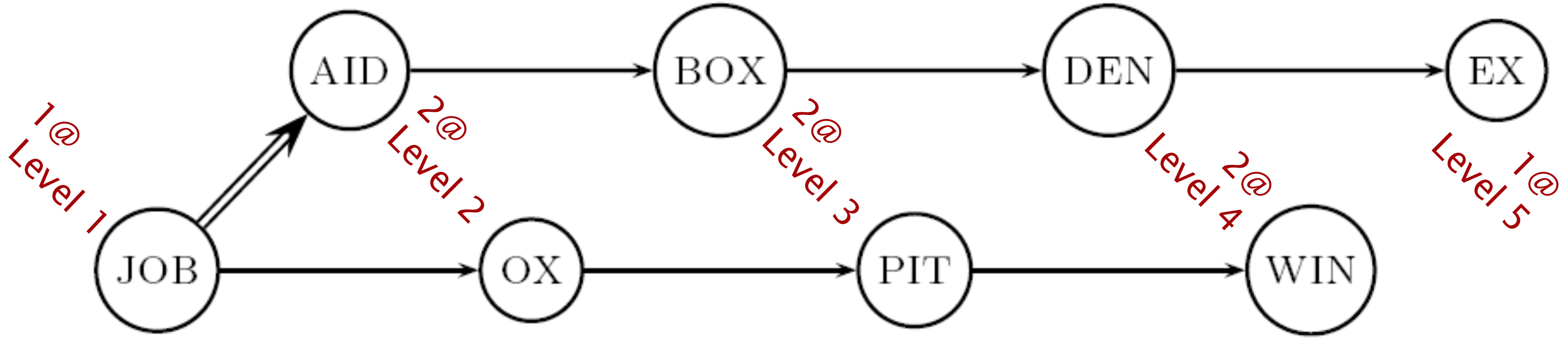
Why not go with a larger *k*?

Dictionary search without blocking

- Assuming each dictionary term equally likely in query (not true in practice!), average number of comparisons = (1 + (2*2) + (4*3) + 4)/8 = ~2.6



Dictionary search with blocking



- Binary search down to 4-term block;
  - Then linear search through terms in block.
- Blocks of 4 (binary tree), average =
  (1 + (2*2) + (2*3) + (2*4) + 5)/8 = 3 compares

Front coding

- Sorted words commonly have long common prefix – store differences only
  - Used in the (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 ***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
Also, blocking *k* = 4	7.1
Also, Blocking + front coding	5.9

POSTINGS COMPRESSION

Postings compression

- The postings file is much larger than the dictionary, factor of at least 10.
- Key desideratum: 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 a lot less than 20 bits per docID.

Postings: two conflicting forces

- A term like ***arachnocentric*** occurs in maybe one doc out of a million – we would like to store this posting using $\log_2$ 1M ~ 20 bits.
- A term like ***the*** occurs in virtually every doc, so 20 bits/posting is too expensive.
  - Prefer 0/1 bitmap vector in this case

Postings file entry

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

Three postings entries

	Encoding	Postings List					
the	docIDs	…	283042	283043	283044	283045	…
	gaps			1	1	1	
computer	docIDs	…	2803047	283154	283159	283202	…
	gaps			107	5	43	
arachno-centric	docIDs	25200	500100				
	gaps	25200	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 ***variable length encoding***
- Variable length codes achieve this by using short codes for small numbers

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 <u>continuation</u> 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

512+256+32+16+8 = 824

Postings stored as the byte concatenation

00000110 10111000 10000101 00001101 00001100 10110001

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

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

Other variable unit codes

- Instead of bytes, we can also use a different "unit of alignment": 32 bits (words), 16 bits, 4 bits (nibbles).
- Variable byte alignment wastes space if you have many small gaps – nibbles do better in such cases.
- Variable byte codes:
  - Used by many commercial/research systems
  - Good blend of variable-length coding and sensitivity to computer memory alignment

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

Summary: Index compression

- We can now create an index for highly efficient Boolean retrieval that is very space efficient
- Use the sorted nature of the data to compress
  - Variable sized storage
  - Encode common prefixes only once
  - Encode gaps to reduce size of numbers
- However, here we didn't encode positional information
  - But techniques for dealing with postings are similar

Resources for today's lecture

- *IIR* 5
- *MG* 3.3, 3.4.
- F. Scholer, H.E. Williams and J. Zobel. 2002. Compression of Inverted Indexes For Fast Query Evaluation. *Proc. ACM-SIGIR 2002*.
  - Variable byte codes
- V. N. Anh and A. Moffat. 2005. Inverted Index Compression Using Word-Aligned Binary Codes. *Information Retrieval* 8: 151–166.
  - Word aligned codes