

Introduction to **Information Retrieval**

Lecture 9: Index Compression

This lecture

| | | | | | | | | | | |
|-----------|---|---|----|----|-----|----|----|-----|-----|-----|
| 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
 - How big are the dictionary and postings likely to be, for a given text documents collection?
- Dictionary compression
- Postings compression

Why compression (in general)?

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

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)
- We will devise various IR-specific compression schemes

Sample text collection: Reuters RCV1

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

Observations

- Preprocessing greatly affects the size of dictionary and number of postings
 - Stemming, case folding, stop word removal
- Percentage reduction can be different based on properties of the collections
 - E.g., lemmatizer for French reduces dictionary size much more than Porter stemmer for English

Index parameters vs. what we index

(details in *IIR book (Manning et al)* Table 5.1)

| size of | word types (terms) | | | non-positional postings | | | positional postings | | |
|---------------|--------------------|------------|---------|-------------------------|------------|---------|---------------------|------------|---------|
| | dictionary | | | non-positional index | | | positional index | | |
| | Size (K) | $\Delta\%$ | cumul % | Size (K) | $\Delta\%$ | cumul % | Size (K) | $\Delta\%$ | 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 |

Exercise: give intuitions for all the '0' entries. Why do some zero entries correspond to big deltas in other columns?

Lossless vs. lossy compression

- Lossless compression: All information is preserved.
- Lossy compression: Discard some information
 - Makes sense when the discarded information is unlikely to be ever used by the IR system
- Several of the preprocessing steps can be viewed as lossy compression: case folding, stop words, stemming, number elimination.

Vocabulary vs. collection size

- How big is the term vocabulary?
 - That is, how many distinct words are likely to be present in a corpus / document collection?
- Can we assume an upper bound?
- In practice, the vocabulary will keep growing with the collection size

Vocabulary vs. collection size

- Heaps' law: $M = kT^b$
- M is the **size of the vocabulary (# distinct terms)**
- T is the total **number of tokens in the collection**
- Typical values: $30 \leq k \leq 100$ and $b \approx 0.5$
- In a log-log plot of vocabulary size M vs. T , **Heaps' law predicts a line with slope about $\frac{1}{2}$**
 - 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

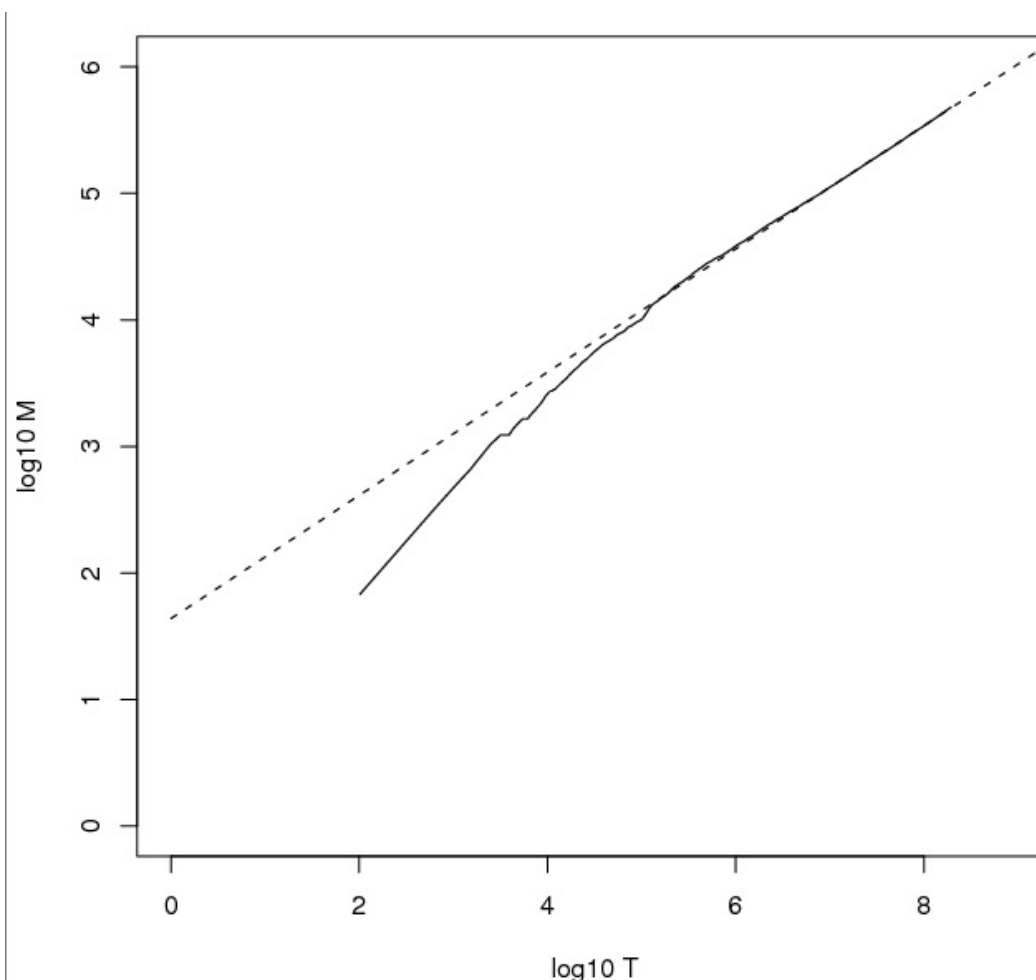
$\log_{10} M = 0.49 \log_{10} T + 1.64$
is the best least squares fit.

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

Fig 5.1 in Manning book



Heap's Law suggests that

- The size of the dictionary is quite large for large collections
- The dictionary continues to increase with more documents in the collection, rather than a maximum vocabulary size being reached

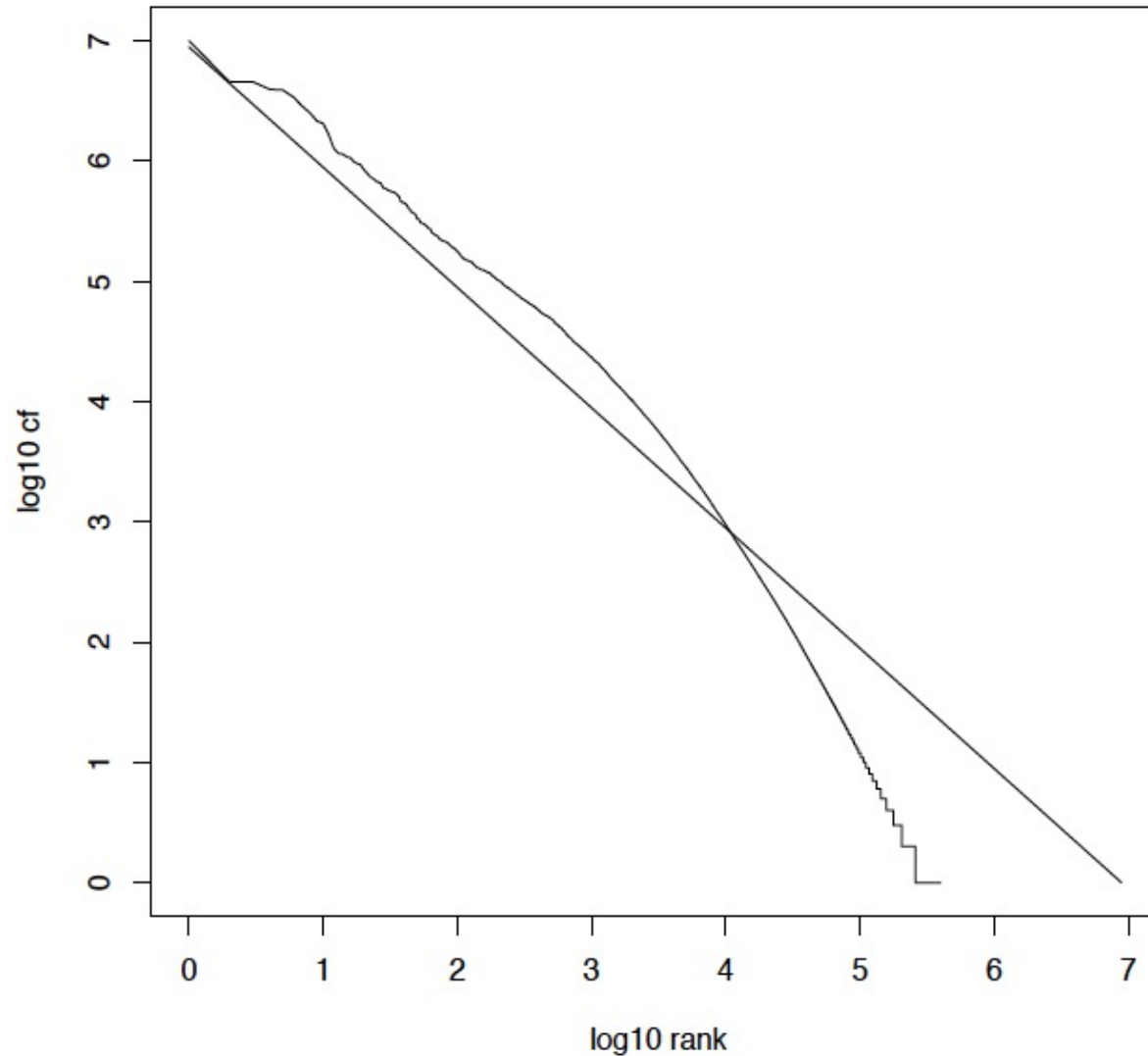
Zipf's law

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

Zipf's Law 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



Compression

- Now, we will consider compressing the space for the dictionary and postings
 - Basic Boolean index only
 - No study of positional indexes, etc.

- We will consider compression schemes
 - Dictionary compression
 - Postings list compression

DICTIONARY COMPRESSION

Why compress the dictionary?

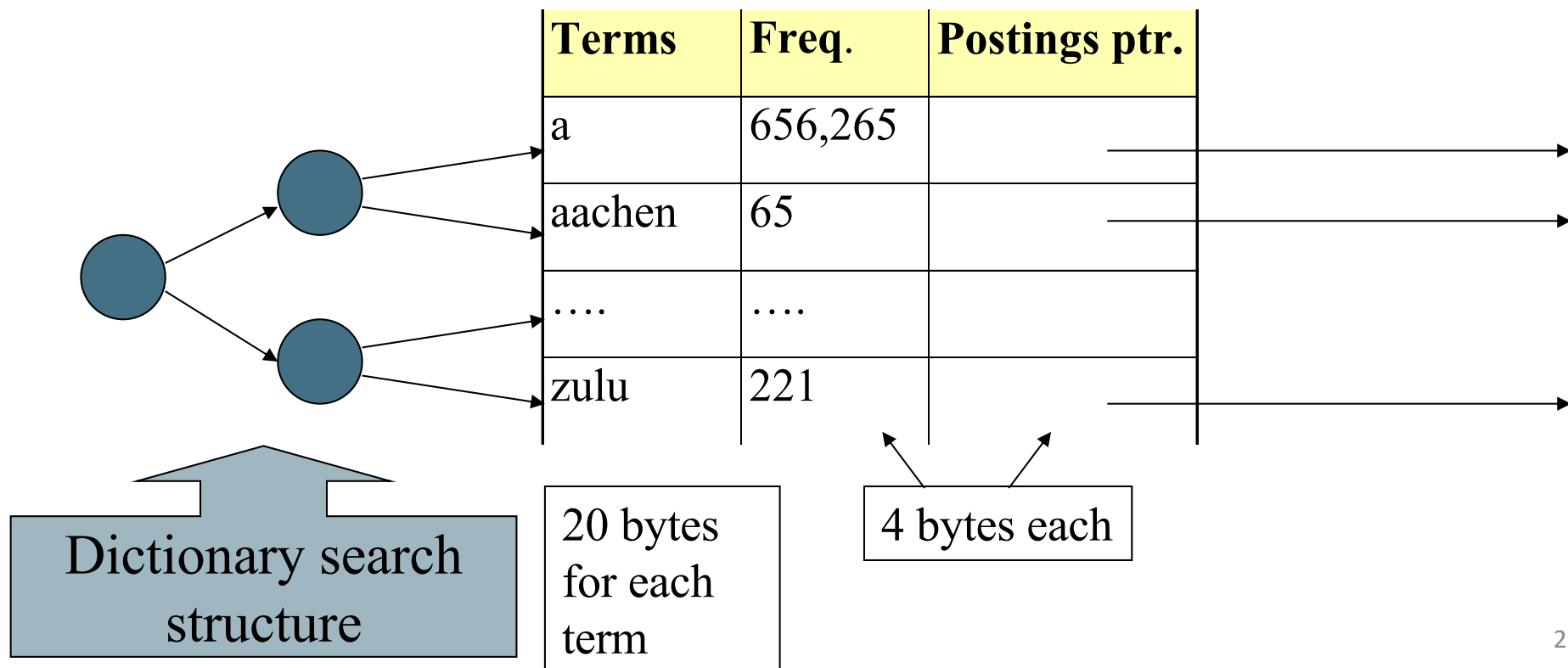
- Search begins with the dictionary
- We want to keep it in memory
- Memory footprint: competition with other applications
- Embedded/mobile devices may have very little memory
- Even if the dictionary isn't in memory, we want it to be small for a fast search startup time
- So, compressing the dictionary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

Sample text collection: Reuters RCV1

| ■ symbol | statistic | value |
|----------|---|-------------|
| ■ N | documents | 800,000 |
| ■ L | avg. # tokens per doc | 200 |
| ■ M | terms (= word types) | ~400,000 |
| ■ | avg. # bytes per token (incl. spaces/punct.) | 6 |
| ■ | avg. # bytes per token (without spaces/punct.) | 4.5 |
| ■ | avg. # bytes per term | 7.5 ~ 8 |
| ■ | non-positional postings | 100,000,000 |

Dictionary storage - first cut

- Array of fixed-width entries (e.g., array of structures)
 - ~400,000 terms; 28 bytes/term = 11.2 MB.



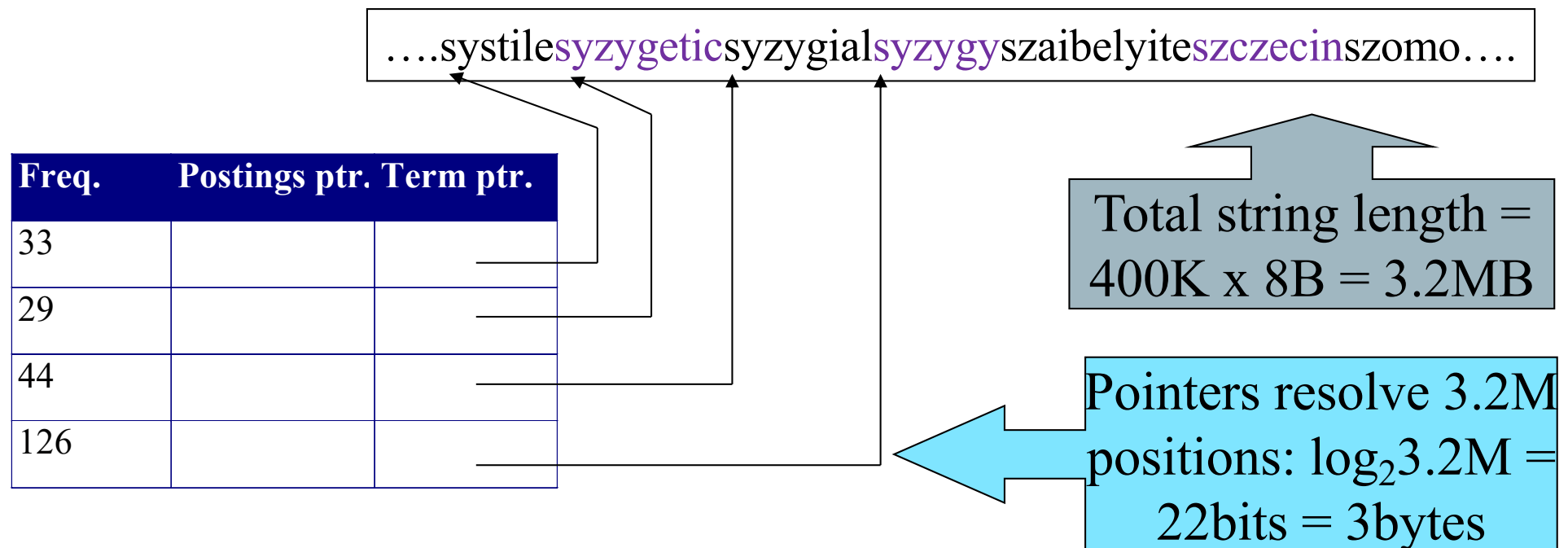
Fixed-width terms are wasteful

- Most of the bytes in the **Term** column are wasted – we allot 20 bytes even for 1 letter terms.
 - And we still can't handle *terms with more than 20 chars*
- Written English averages ~4.5 characters/word
- Ave. dictionary word in English: ~8 characters

Compressing the term list:

Approach 1: 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.

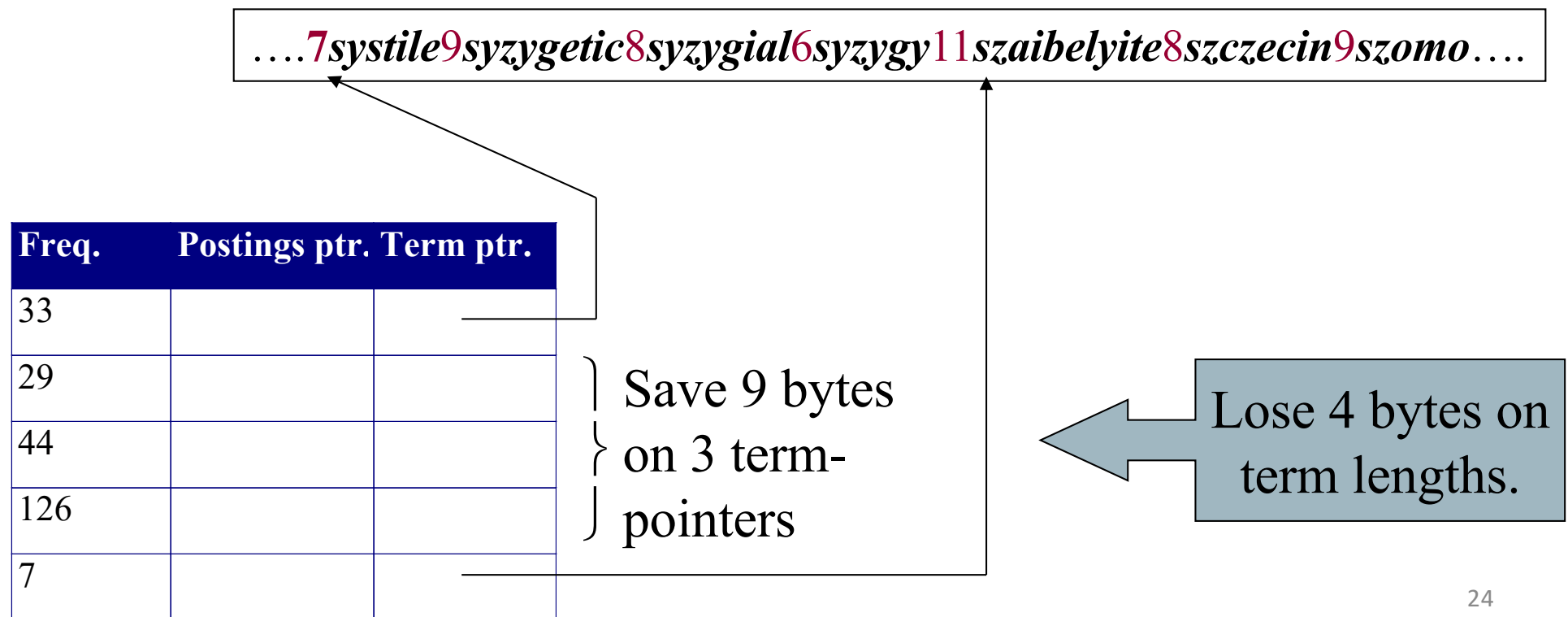


Space for dictionary as a string

- 4 bytes per term for Freq.
 - 4 bytes per term for pointer to Postings.
 - 3 bytes per term pointer
 - Avg. 8 bytes per term in term string
 - 400K terms $\times 19 \Rightarrow 7.6$ MB (against 11.2MB for fixed width)
- } Now avg. 11 bytes/term, not 20.

Approach 2: Blocking

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



Blocking

- Group terms into blocks, each having k terms
- Store a term pointer only for first term of each block
- Store the length of each term as one additional byte at the beginning of each term
- Search for terms in the compressed dictionary
 - Locate the term's block by **binary search**
 - Then locate term's position within the block by **linear search within the block**
- By increasing block size k : tradeoff between better compression and speed of term lookup

Net saving

- Example for block size $k = 4$
- Where we used 3 bytes/pointer without blocking
 - $3 \times 4 = 12$ bytes,

now we use $3 + 4 = 7$ bytes.

Saved another ~ 0.5 MB. This reduces the size of the dictionary from 7.6 MB to 7.1 MB.

We can save more with larger k .

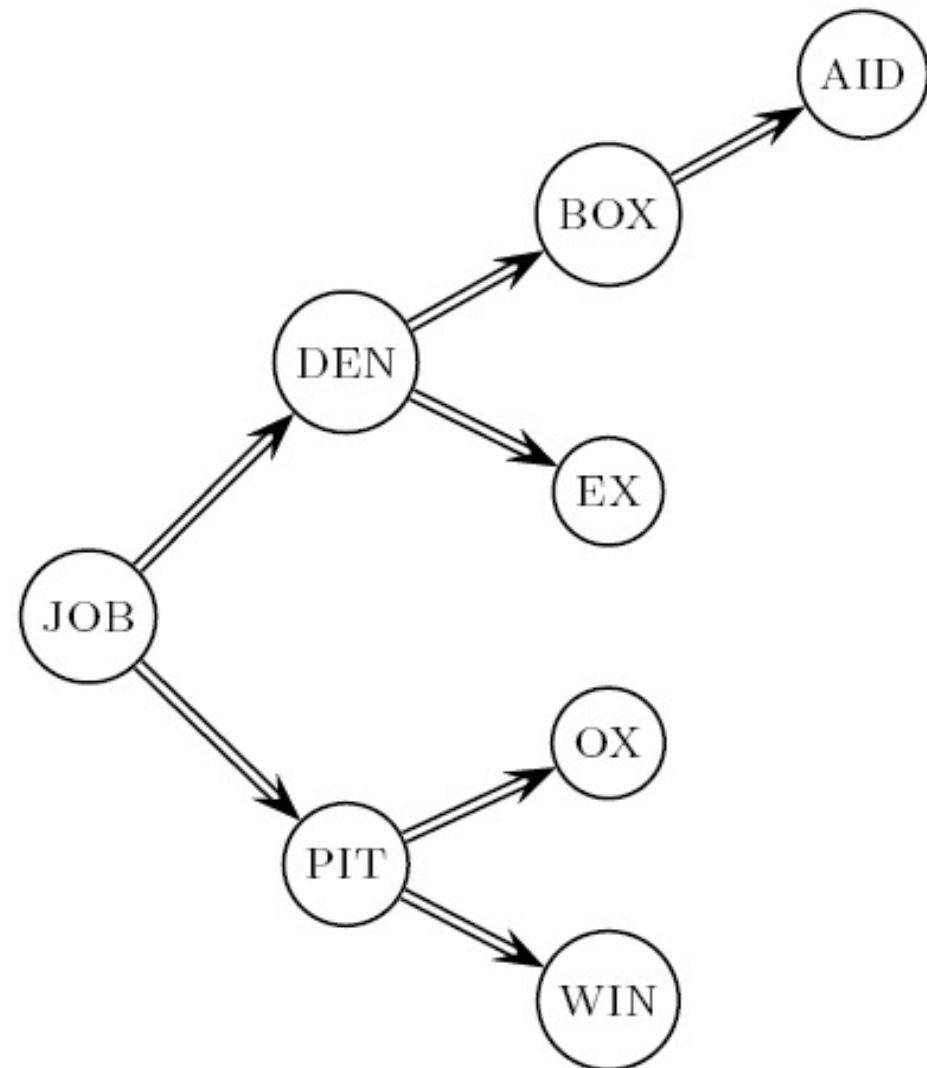
Why not go with larger block size k ?

Tradeoff around value of k

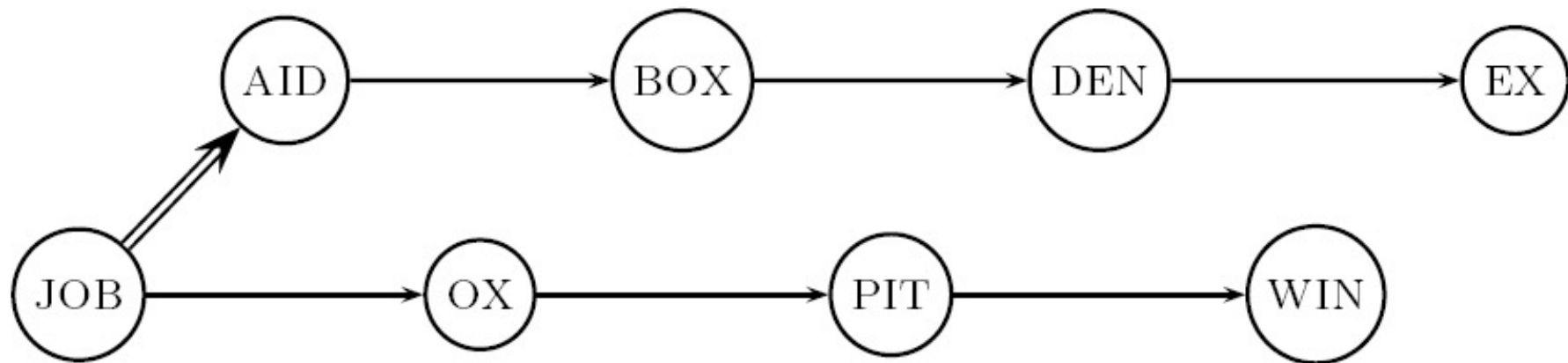
- By increasing block size k
 - Better compression, i.e., less space needed
 - But, more time required for term lookup
- How do we search for a term?
 - In an uncompressed dictionary – by binary search
 - In a compressed dictionary – locate the term's block by binary search, then find its position within the block by linear search
- Block size k controls the time taken by binary search vs. linear search

Dictionary search without blocking

- Assuming each dictionary term equally likely in query (not really so in practice!), average number of comparisons = $(1+2\cdot 2+4\cdot 3+4)/8 \sim 2.6$



Dictionary search with blocking



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

Exercise

- Estimate the impact on search performance (and slowdown compared to $k=1$) with blocking, for block sizes of $k = 4, 8$ and 16 .
- If the frequencies of query terms were non-uniform but known, how would you structure the dictionary search tree?

Approach 3: 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 *automat*

Extra length
beyond *automat*.

Begins to resemble general string compression. 31

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 far fewer 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 $\log_2 1M \sim 20$ bits.
- A term like ***the*** occurs in virtually every doc, so 20 bits/posting is too expensive.

Postings file entry

- We store the list of docs containing a term in increasing order of docID.
 - **computer**: 33,47,154,159,202 ...
- Consequence: it suffices to **store gaps**.
 - 33,14,107,5,43 ...
- Hope: 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 | ... | 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 $\sim \log_2 G$ 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*

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 $G \leq 127$, 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 (left-most bit is continuation bit)

| 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.

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 low-tech blend of variable-length coding and sensitivity to computer memory alignment matches (vs. bit-level codes, which we look at next).

Variable bit-level codes: Unary code

- Represent n as n 1s with a final 0.
- Unary code for 3 is 1110.

- Unary code for 40 is

11111111111111111111111111111111111110 .

- Unary code for 80 is:

11

110

- This doesn't look promising, but....

Gamma codes

- We can compress better with bit-level codes
 - The Gamma code is the best known of these.
- Represent a gap G as a pair *length* and *offset*
- *offset* is G in binary, with the leading bit cut off
 - For example $13 \rightarrow 1101 \rightarrow 101$
- *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 |

Gamma code properties

- G is encoded using $2 \lfloor \log G \rfloor + 1$ bits
 - Length of offset is $\lfloor \log G \rfloor$ bits
 - Length of length is $\lfloor \log G \rfloor + 1$ bits
- All gamma codes have an odd number of bits
- Almost within a factor of 2 of best possible, $\log_2 G$
- Gamma code is uniquely prefix-decodable, like VB
- Gamma code can be used for any distribution
- Gamma code is parameter-free

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 |

Index compression summary

- We can now create an index for highly efficient Boolean retrieval that is very space efficient
- Only 4% of the total size of the collection
- Only 10-15% of the total size of the text in the collection
- However, we've ignored positional information
- Hence, space savings are less for indexes used in practice
 - But techniques substantially the same.