# Index compression

## CE-324: Modern Information Retrieval

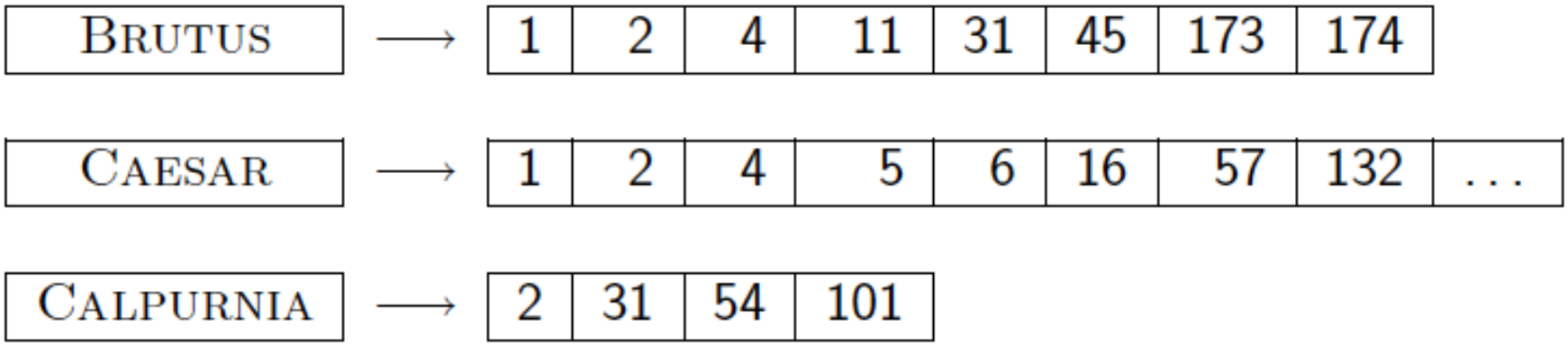
Sharif University of Technology

M. Soleymani

Fall 2018

Most slides have been adapted from: Profs. Manning, Nayak & Raghavan (CS-276, Stanford)

# Today




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

# Why compression (in general)?

- Use less disk space
  - Saves a little money
- Keep more stuff 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

# 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

# Compression

- Compressing the space for the dictionary and postings
  - Basic Boolean index only
  - No study of positional indexes, etc.
  - We will consider compression schemes

# Reuters RCV1 statistics

| symbol | statistic | value |
|---|---|---|
| N | # documents | 800,000 |
| $L_{ave}$ | 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 |

# Index parameters vs. what we index
## (details IIR Table 5.1, p.80)

| | Dictionary (terms) | | | non-positional postings | | | positional postings | | |
|---|---|---|---|---|---|---|---|---|---|
| | Size (K) | Δ% | Total % | Size (K) | Δ% | Total % | Size (K) | Δ% | Total% |
| 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.
  - 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.

- 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
- 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

# Main goal of dictionary compression

- Fit it (or at least a large portion of it) in main memory
  - to support high query throughput

# Vocabulary 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 Unicode ☺

# Vocabulary vs. collection size

- **Heaps' law**: $M = kT^b$
  - *M*: # terms
  - *T* : # tokens
  - Typical values: 30 ≤ *k* ≤ 100 and *b* ≈ 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

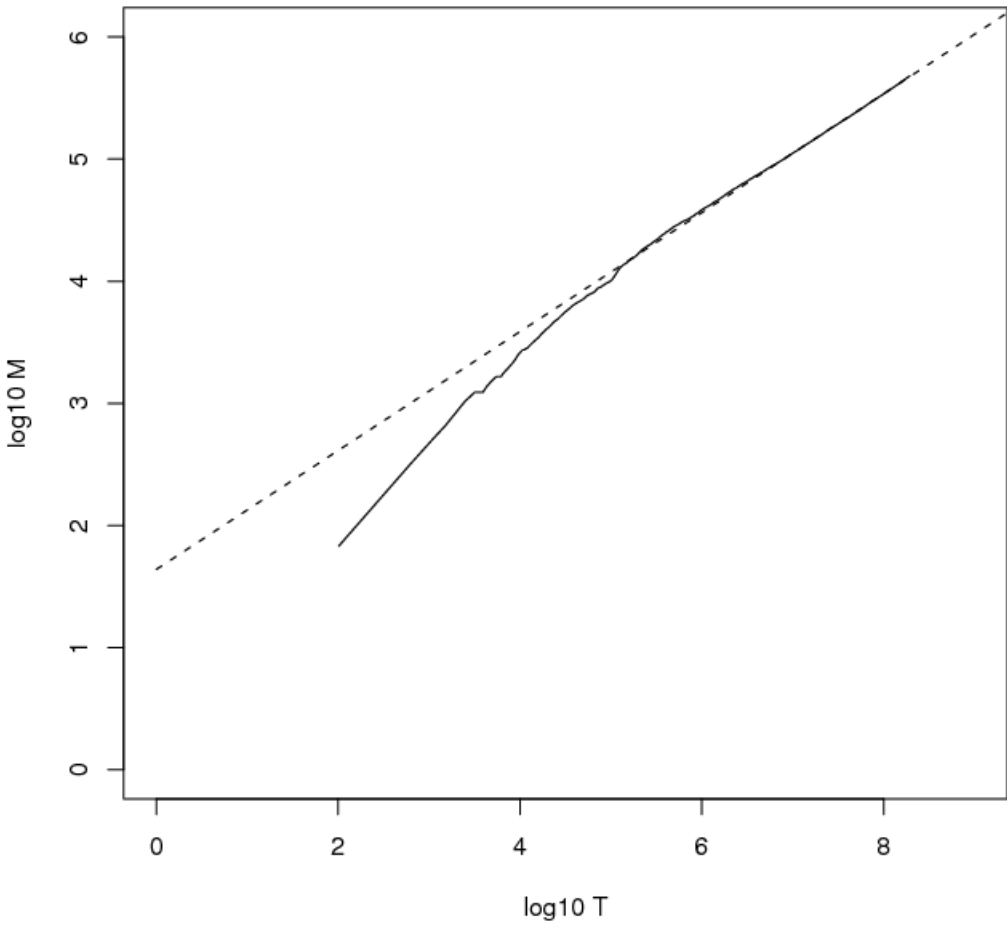
$M = kT^b$

- RCV1:
  - $M = 10^{1.64}T^{0.49}$
  - $k = 10^{1.64} \approx 44$
  - $b = 0.49$.

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

(best least squares fit)

For first 1,000,020 tokens, predicts 38,323 terms; actually, 38,365 terms




Good empirical fit for Reuters RCV1 !

# A naïve dictionary

- An array of struct:

| term | document frequency | pointer to postings list |
|---|---|---|
| a | 656,265 | ⟶ |
| aachen | 65 | ⟶ |
| . . . | . . . | . . . |
| zulu | 221 | ⟶ |

20 bytes 4/8 bytes 4/8 bytes

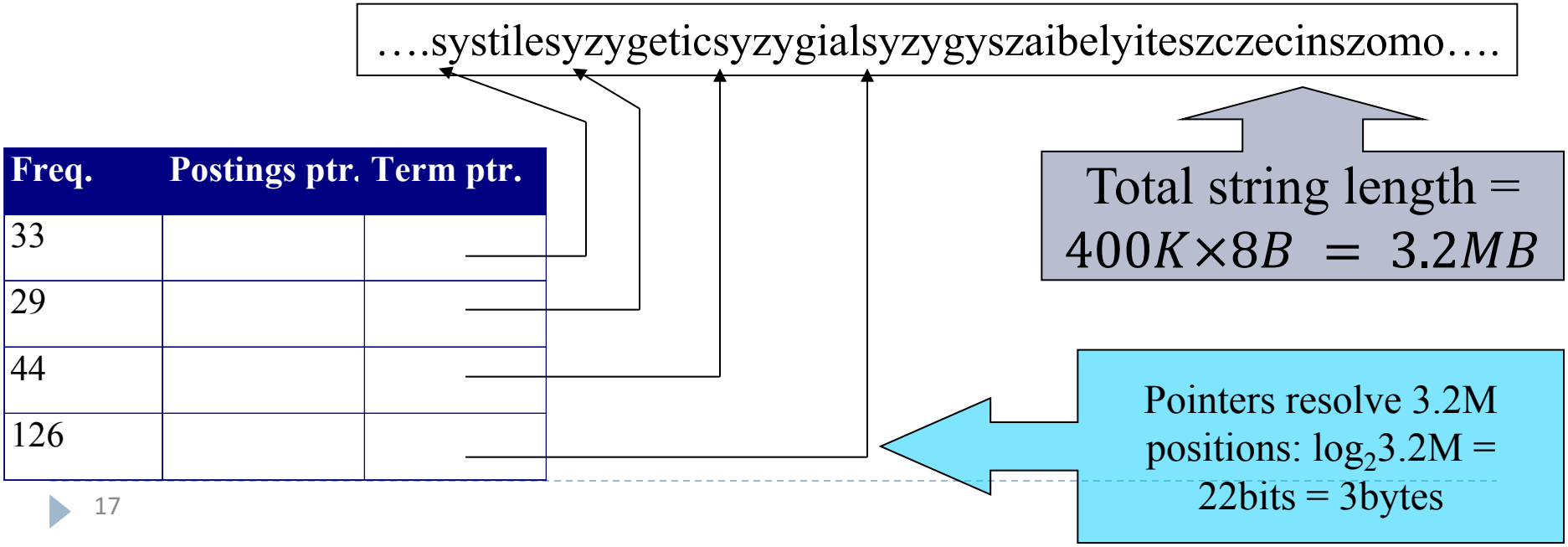
- How do we store a dictionary in memory efficiently?
- How do we quickly look up elements at query time?

# Fixed-width terms are wasteful

- Most of the bytes in the **Term** column are wasted.
  - We allow 20 bytes for 1 letter terms
  - Also we still can't handle *supercalifragilisticexpialidocious* 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.



# Space for dictionary as a string

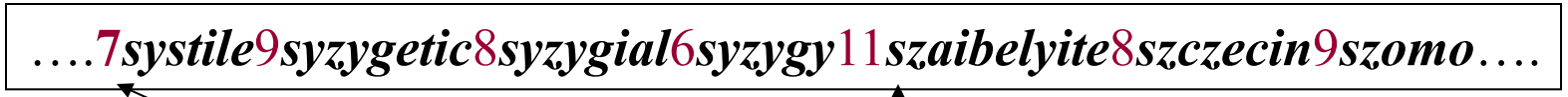
- 4 bytes per term for Freq.
- 4 bytes per term for pointer to Postings.
- 3 bytes per term pointer
- Avg. 8 bytes per term in term string

Now avg. 11 bytes/term, not 20.

- 400K terms x 19 $\Rightarrow$ 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)




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

Save 9 bytes on 3 pointers.

Lose 4 bytes on term lengths.

# Blocking

- Example for block size *k* = 4
- Without blocking: 3 x 4 = 12 bytes
  - Where we used 3 bytes/pointer without blocking
- Blocking: 3 + 4 = 7 bytes.
- Size of the dictionary from 7.6 MB to 7.1 MB (Saved ~0.5MB).

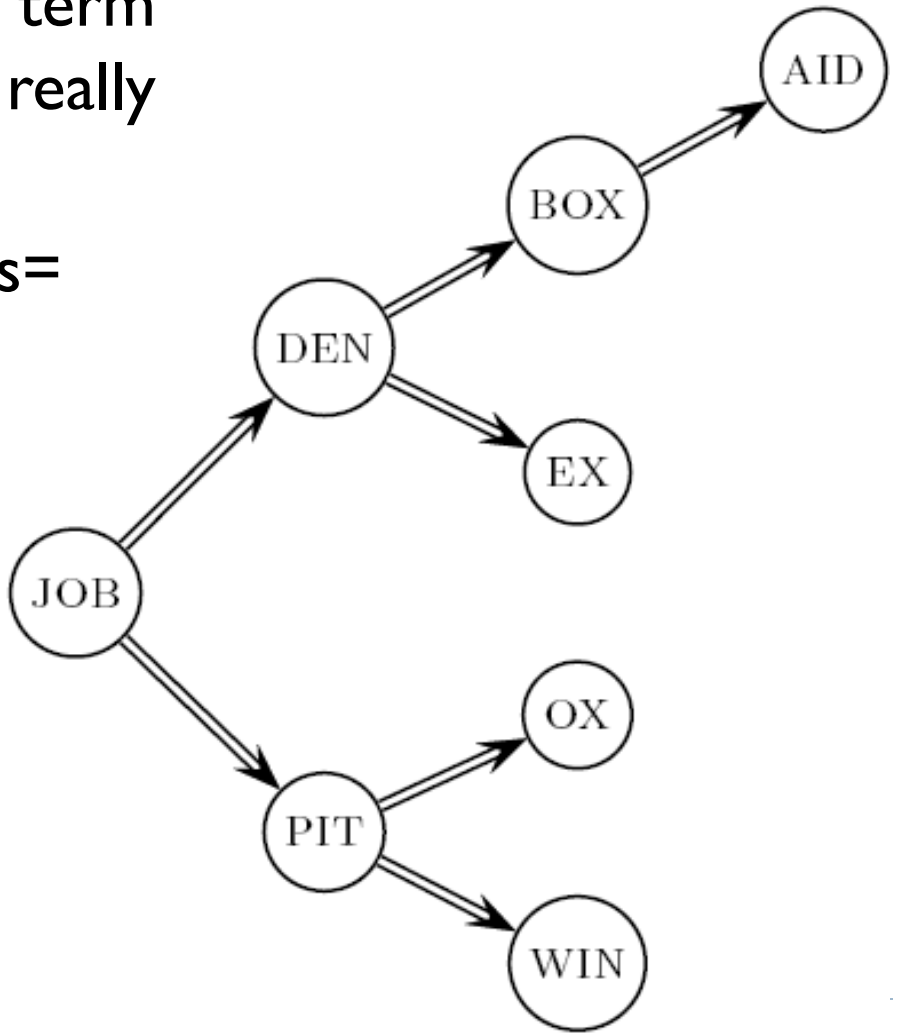
Why not go with larger *k*?

# Dictionary search without blocking

- Assuming each dictionary term equally likely in query (not really so in practice!):

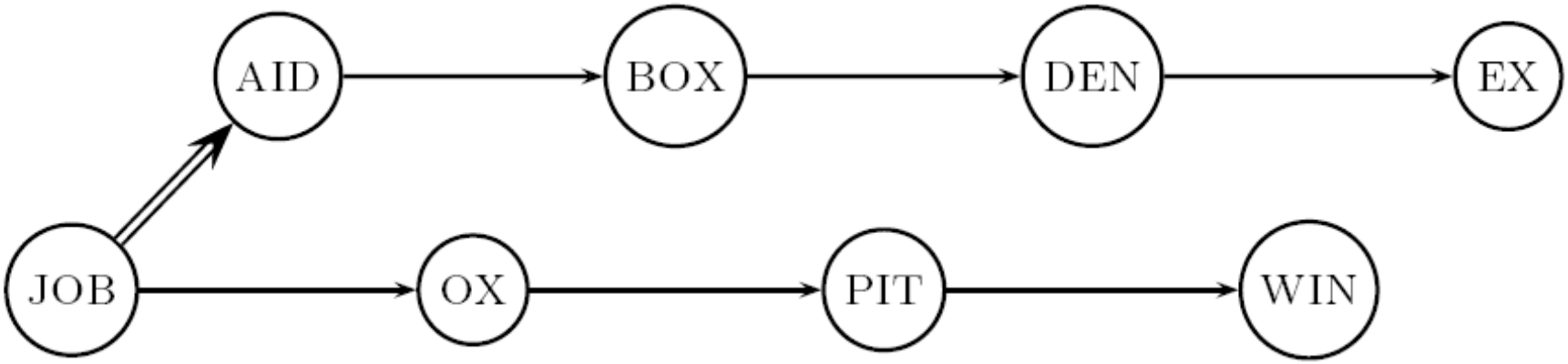
  average no. of comparisons=

  (1+2·2+4·3+4)/8 ~2.6




Exercise: what if the frequencies of query terms were non-uniform but known, how would you structure the dictionary search tree?

# 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+2·2+2·3+2·4+5)/8 = 3 compares

# Front coding

- Front-coding:
  - Sorted words commonly have long common prefix
    - store differences only (for last *k*-*l* 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 docs), we would use 32 bits (4 bytes) per docID when using 4-byte integers.
  - Alternatively, we can use $\log_2$ 800,000 ≈ 20 bits per docID.
- Our goal: use far fewer than 20 bits per docID.

# Postings: two conflicting forces

- ***arachnocentric*** occurs in maybe one doc
  - we would like to store this posting using $\log_2$ 1M ~ 20 bits.

- ***the*** occurs in virtually every doc
  - 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***: 33,47,154,159,202 …
- 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 | ... |
| COMPUTER | docIDs | ... | 283047 | 283154 | 283159 | 283202 | ... |
| ARACHNOCENTRIC | docIDs | 252000 | 500100 | | | | |

# Term frequencies

- 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.

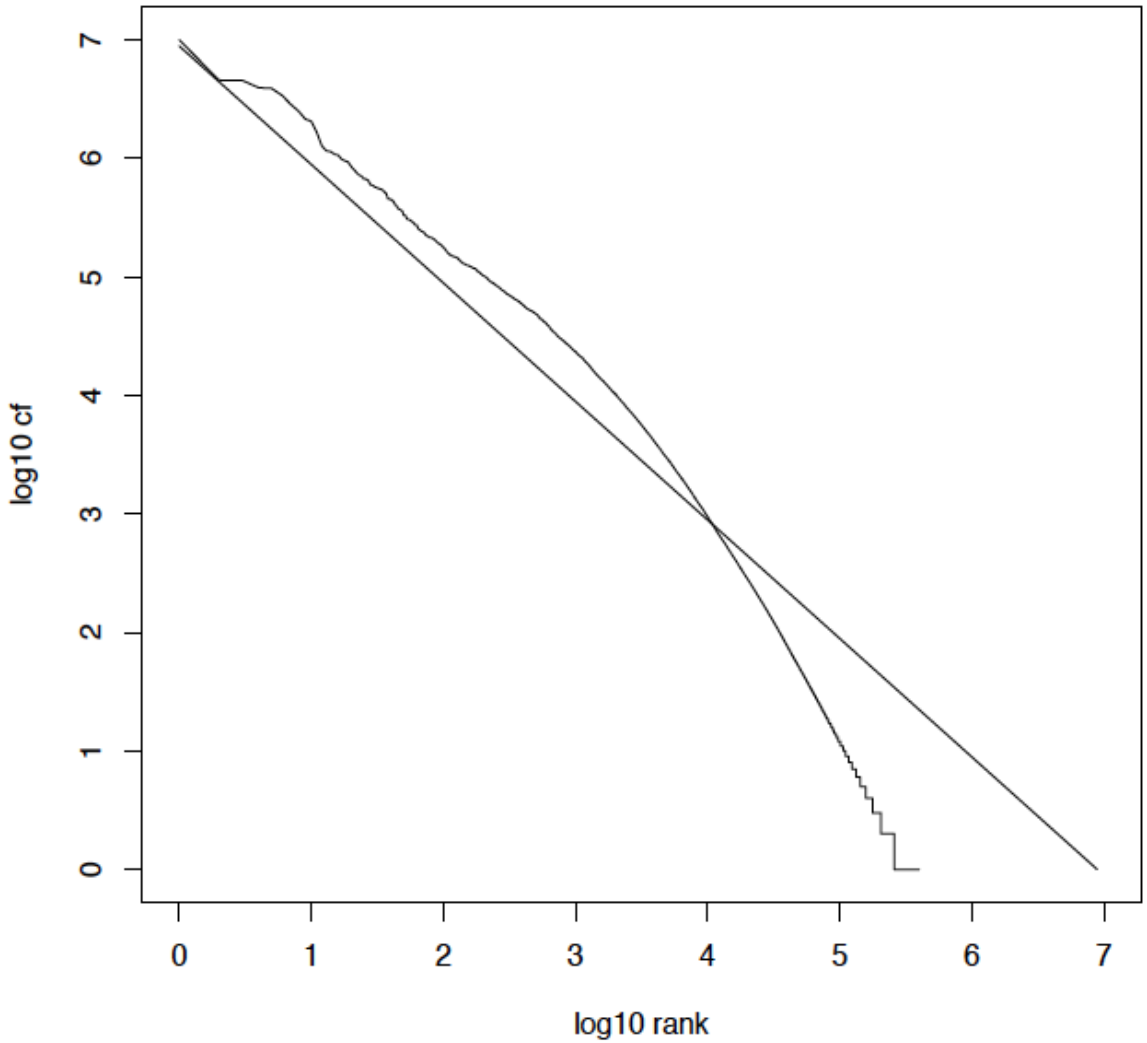
# Zipf's law

- Zipf's law: The *i*th most frequent term has frequency proportional to 1/*i* .
- $cf_i$ is collection frequency: the number of occurrences of the term $t_i$ in the collection.

# Zipf consequences

- Most frequent term occurs $\underline{cf_1}$ times
  - second most frequent term occurs $\underline{cf_1/2}$ times
  - third most frequent term occurs $\underline{cf_1/3}$ times …

# Zipf's law for Reuters RCV1




$$cf_i \propto \frac{1}{i}$$

# Variable length encoding

- Average gap for a term: *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.
  - For a gap value *G*, we want to use close to $\log_2 G$ bits

- This requires a *variable length encoding*
  - using short codes for small numbers

# Variable Byte (VB) codes

- 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
  - Else encode *G*'s lower-order 7 bits and then use additional bytes to encode the higher order bits recursively
  - At the end: set the continuation bits
    - the last byte *c* =1
    - other bytes *c* = 0.

# Example

| docIDs | 824 | 829 | 215406 |
|---|---|---|---|
| gaps | | 5 | 214577 |
| VB code | 00000110<br>10111000 | 10000101 | 00001101<br>00001100<br>10110001 |

Postings stored as the byte concatenation

000000110101110001000010100001101000011001010110001

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

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

# Other variable unit codes

- Other "unit of alignment" instead of bytes:
  - 32 bits (words), 16 bits, 4 bits (nibbles).
  - Variable byte may waste space when many small gaps (nibbles do better)

- 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

# Unary code

- Represent *n*: *n* 1s + a 0
  - 3:1110
  - 38:11111111111111111111111111111111111110

- This doesn't look promising, but….

# Gamma codes

- We can compress better with bit-level codes
  - Gamma code: the best known bit-level.

- Represent a gap *G*: *length* + *offset*
  - *Offset*: *G* in binary, with the leading bit cut off
    - E.g., 13 → 1101 → 101
  - *Length*: length of offset encoded with *unary code*
    - E.g., 13 (offset 101), length is 3 → 1110.
  - Gamma code: *length* + *offset*
    - E.g., 13 → 1110101

# Gamma code examples

| number | length | offset | g-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 | 1111111110 | 0000000001 | 1111111110,0000000001 |

# Gamma code properties

- $G \rightarrow 2\lfloor \log G \rfloor + 1$ bits
  - Offset: $\lfloor \log G \rfloor$ bits
  - Length: $\lfloor \log G \rfloor + 1$ bits

- Properties:
  - always have an odd number of bits
  - almost within a factor of 2 of best possible ($\log_2 G$)
  - uniquely prefix-decodable, like VB
  - can be used for any distribution
  - parameter-free

# Gamma seldom used in practice

- Machines have word boundaries (8, 16, 32, 64 bits)
  - Operations that cross word boundaries are slower
  - Compressing and manipulating at the granularity of bits can be slow
- Variable byte encoding is aligned and thus potentially more efficient
- Regardless of efficiency, variable byte is conceptually simpler at little additional space cost

# 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 |
| collection (text) | 960 |
| Term-doc incidence matrix | 40,000 |
| postings, uncompressed (32-bit words) | 400 |
| postings, uncompressed (20 bits) | 250 |
| postings, variable byte encoded | 116 |
| postings, g-encoded | 101 |

# 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.