Introduction to

Information Retrieval

Michel Schellekens

Adapted from slides by:

Hinrich Schütze and Christina Lioma

Lecture 5: Index Compression

Outline

Why compression? (in general)

- Use less disk space (saves money)
- Keep more stuff in memory (increases speed)
- Increase speed of transferring data from disk to memory (again, increases speed)
  - [read compressed data and decompress in memory]
    is faster than
    [read uncompressed data]
- Premise: Decompression algorithms are fast.
- This is true of the decompression algorithms we will use.

Why compression in information retrieval?

- First, we will consider space for dictionary
  - Main motivation for dictionary compression: make it small enough to keep in main memory
- Then for the postings file
  - Motivation: reduce disk space needed, decrease time needed to read from disk
  - Note: Large search engines keep significant part of postings in memory
- We will devise various compression schemes for dictionary and postings.

Lossy vs. lossless compression

- Lossy compression: Discard some information
- Several of the preprocessing steps we frequently use can be viewed as lossy compression:
  - downcasing, stop words, porter, number elimination
- Lossless compression: All information is preserved.
  - What we mostly do in index compression

Outline

Model collection: The Reuters collection

symbol	statistics	value
N	documents	800,000
L	avg. # tokens per document	200
M	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 (= word type)	7.5
T	non-positional postings	100,000,000

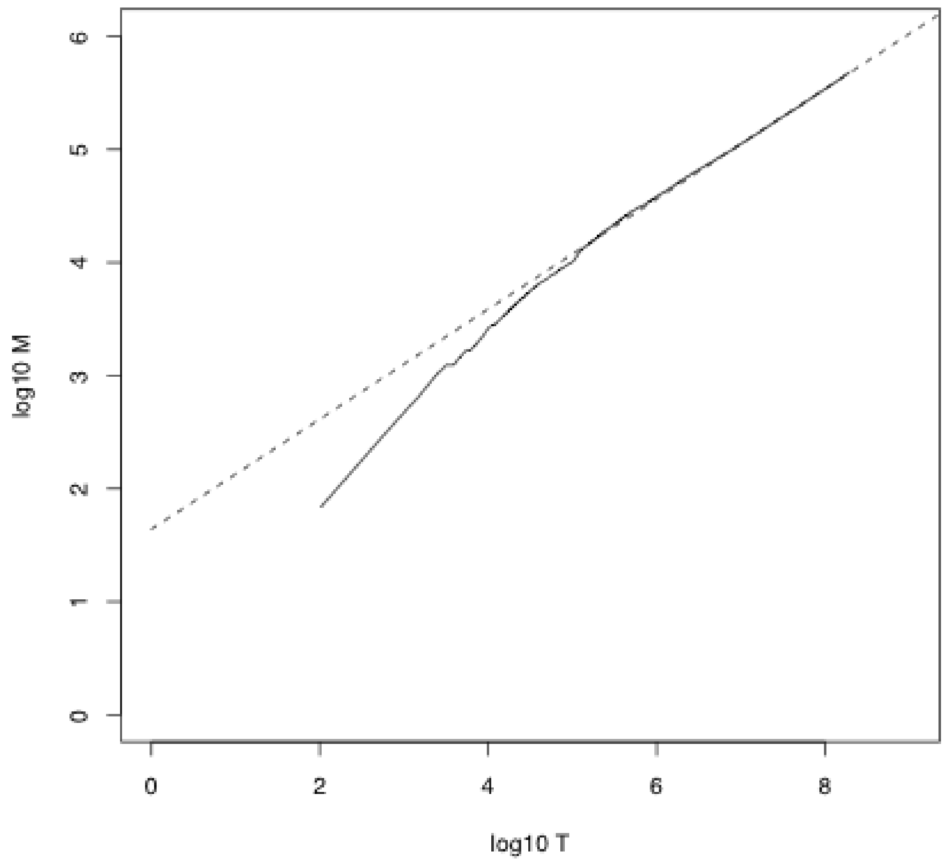
Effect of preprocessing for Reuters

size of	word types (term)			non-positional postings			positional postings (word tokens)		
	dictionary			non-positional index			positional index		
	size	Δ	cml..	size	Δ	cml..	size	Δ	cml..
unfiltered	484,494			109,971,179			197,879,290		
no numbers	473,723	-2%	-2%	100,680,242	-8%	-8%	179,158,204	-9%	-9%
case folding	391,523	-17%	-19%	96,969,056	-3%	-12%	179,158,204	-0%	-9%
30 stop w's	391,493	-0%	-19%	83,390,443	-14%	-24%	121,857,825	-31%	-38%
150 stop w's	391,373	-0%	-19%	67,001,847	-30%	-39%	94,516,599	-47%	-52%
stemming	322,383	-17%	-33%	63,812,300	-4%	-42%	94,516,599	-0%	-52%

How big is the term vocabulary?

- That is, how many distinct words are there?
- Can we assume there is an upper bound?
- Not really: At least $70^{20} \approx 10^{37}$ different words of length 20.
- The vocabulary will keep growing with 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 for the parameters k and b are: $30 \leq k \leq 100$ and $b \approx 0.5$.
- Heaps' law is linear in log-log space.
  - It is the simplest possible relationship between collection size and vocabulary size in log-log space.
  - Empirical law

Heaps' law for Reuters



K = 44 and b = 0.49

$M = kT^b$

Vocabulary size M as a function of collection size T (number of tokens) for Reuters-RCV1The dashed line $\log_{10} M = 0.49 * \log_{10} T + 1.64$ is the best least squares fit. Thus, $M = 10^{1.64} T^{0.49}$ and $k = 10^{1.64} \approx 44$ and $b = 0.49$.

Empirical fit for Reuters

- Good, as we just saw in the graph.
- Example: for the first 1,000,020 tokens Heaps' law predicts 38,323 terms:

$$44 \times 1{,}000{,}020^{0.49} \approx 38{,}323$$

- The actual number is 38,365 terms, very close to the prediction.
- Empirical observation: fit is good in general.

Exercise

Compute vocabulary size *M*

- Looking at a collection of web pages, you find that there are 3000 different terms in the first 10,000 tokens and 30,000 different terms in the first 1,000,000 tokens.
- Assume a search engine indexes a total of 20,000,000,000 (2×10^{10}) pages, containing 200 tokens on average
- What is the size of the vocabulary of the indexed collection as predicted by Heaps' law?

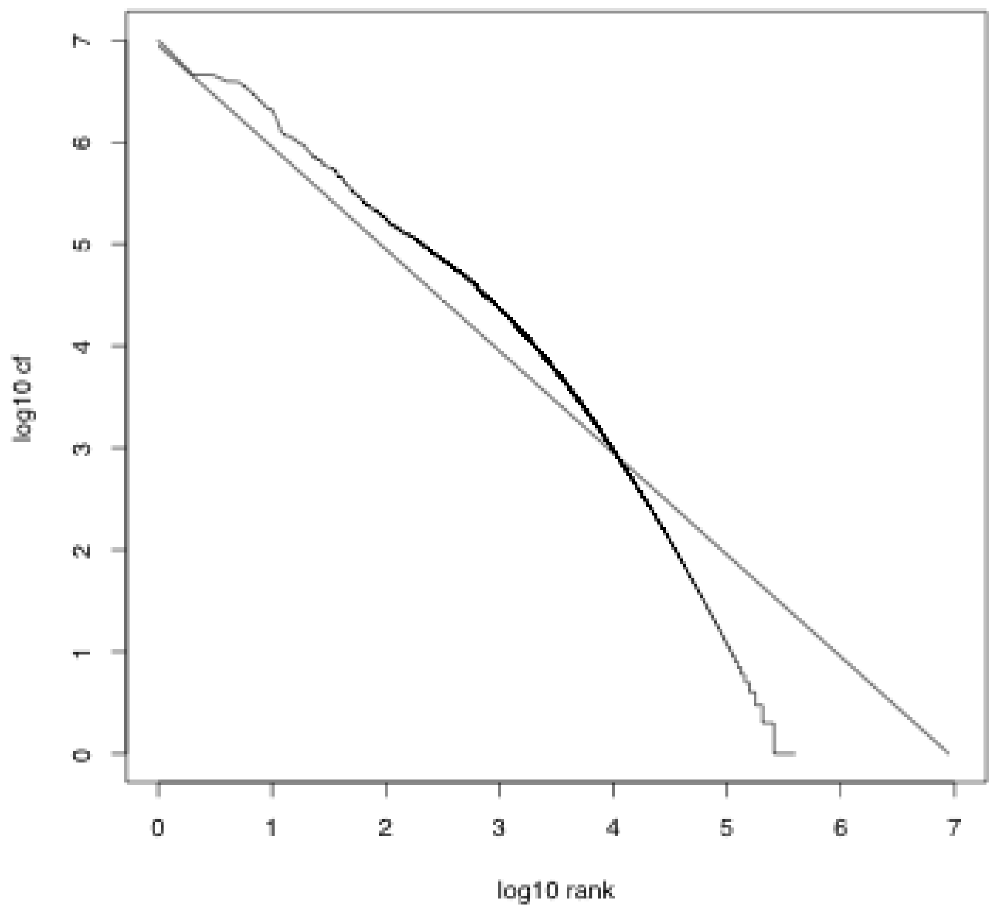
Zipf's law

- Now we have characterized the growth of the vocabulary in collections.
- We also want to know how many frequent vs. infrequent terms we should expect in a collection.
- 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 cf_i proportional to 1/*i* .
- $\mathrm{cf}_i \propto \frac{1}{i}$
- cf_i is collection frequency: the number of occurrences of the term t_i in the collection.

Zipf's law

- Zipf's law: The *i*th most frequent term has frequency proportional to 1/*i* .
- $\mathrm{cf}_i \propto \frac{1}{i}$
- cf is collection frequency: the number of occurrences of the term in the collection.
- So if the most frequent term (*the*) occurs cf_1 times, then the second most frequent term (*of*) has half as many occurrences
  $\mathrm{cf}_2 = \frac{1}{2}\mathrm{cf}_1 \quad \ldots$
- . . . and the t   third most frequent term (*and*) has a third as many occurrences

Zipf's law for Reuters



Fit is not great. What is important is the key insight: Few frequent terms, many rare terms.

Outline

Dictionary compression

- The dictionary is small compared to the postings file.
- But we want to keep it in memory.
- Also: competition with other applications, cell phones, onboard computers, fast startup time
- So compressing the dictionary is important.

Recall: Dictionary as array of fixed-width entries

term	document frequency	pointer to postings list
a	656,265	$\longrightarrow$
aachen	65	$\longrightarrow$
...	...	...
zulu	221	$\longrightarrow$

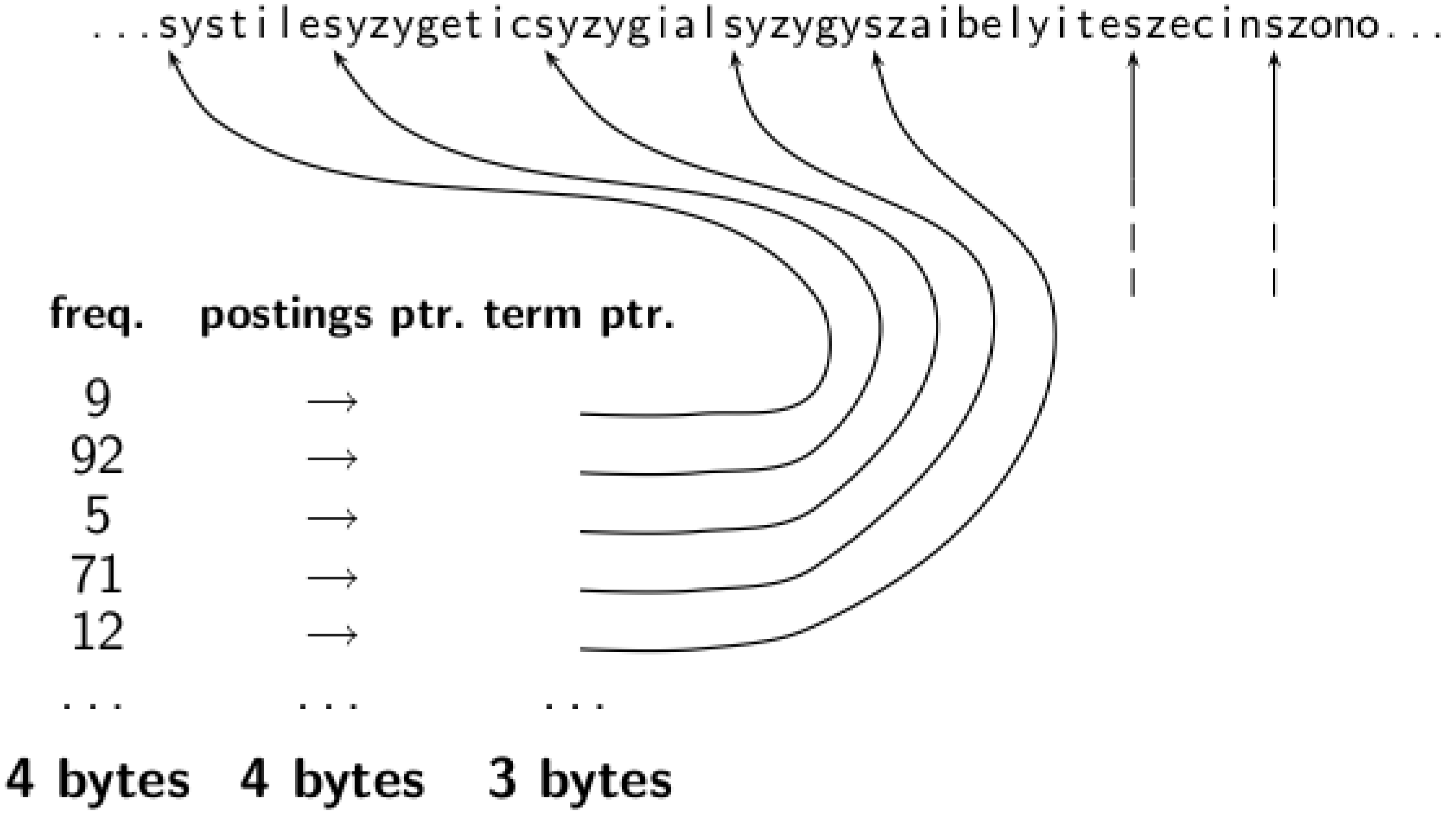
Space needed: 20 bytes 4 bytes 4 bytes

for Reuters: (20+4+4)*400,000 = 11.2 MB

Fixed-width entries are bad.

- Most of the bytes in the term column are wasted.
  - We allot 20 bytes for terms of length 1.
- We can’t handle HYDROCHLOROFLUOROCARBONS and SUPERCALIFRAGILISTICEXPIALIDOCIOUS
- Average length of a term in English: 8 characters
- How can we use on average 8 characters per term?

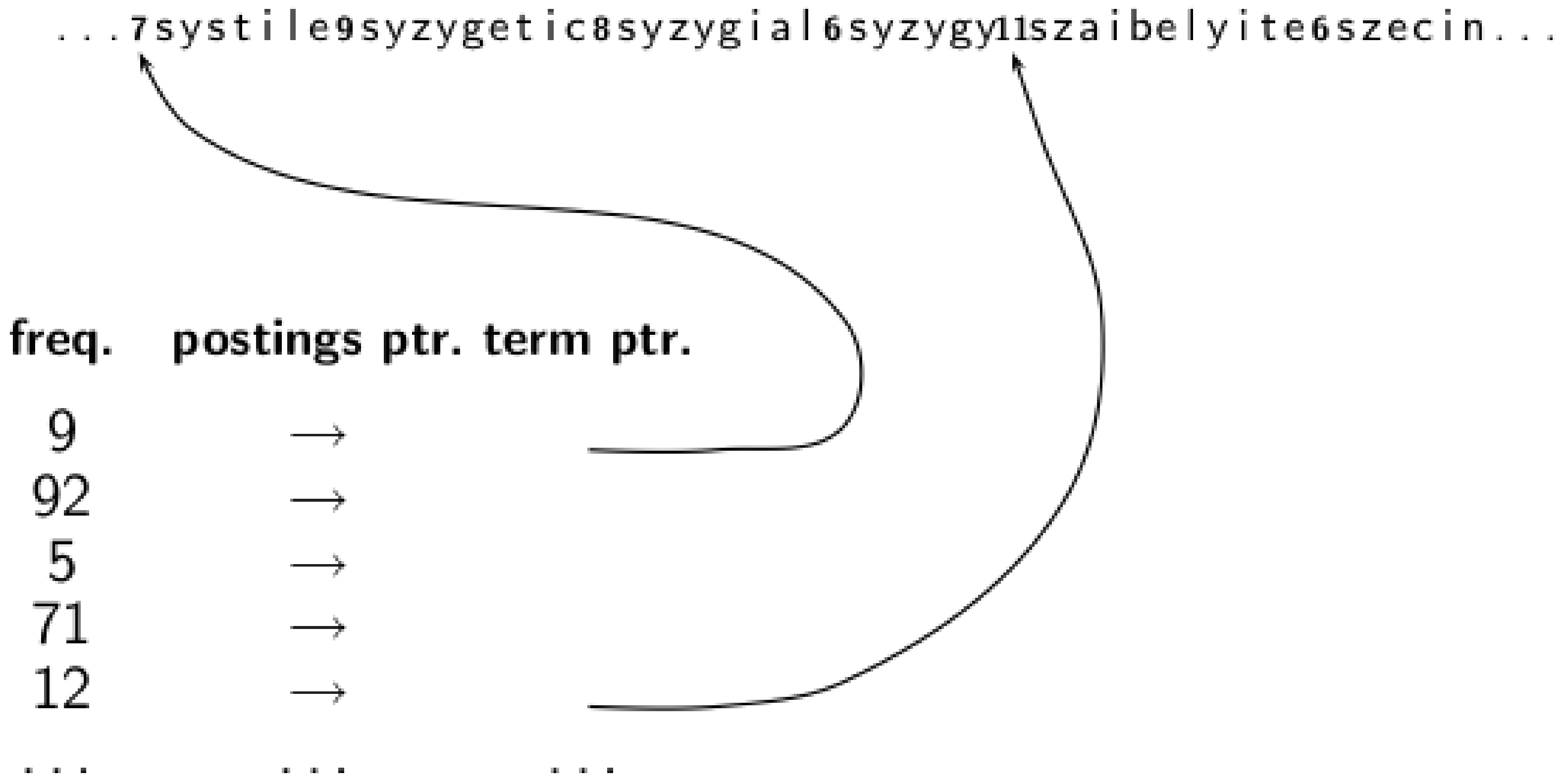
Dictionary as a string



Space for dictionary as a string

- 4 bytes per term for frequency
- 4 bytes per term for pointer to postings list
- 8 bytes (on average) for term in string
- 3 bytes per pointer into string (need $\log_2 8 \cdot 400000 < 24$ bits to resolve $8 \cdot 400{,}000$ positions)
- Space: $400{,}000 \times (4 + 4 + 3 + 8) = 7.6$MB (compared to 11.2 MB for fixed-width array)

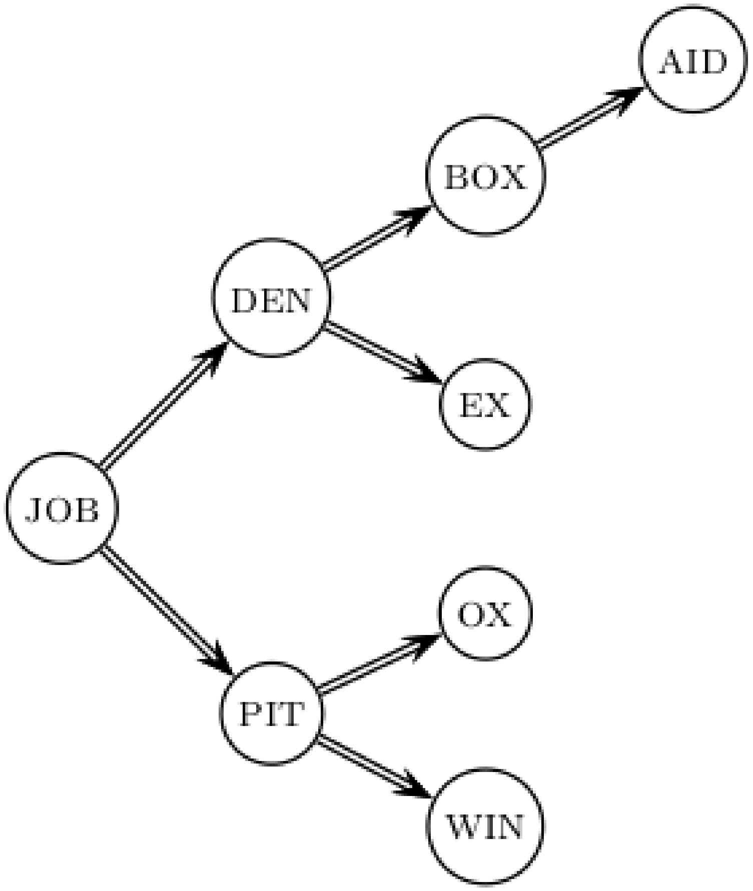
Dictionary as a string with blocking


...7systile9syzygetic8syzygial6syzygy11szaibelyite6szecin...
freq.
postings ptr.
term ptr.
9
92
5
71
12
...
...
...

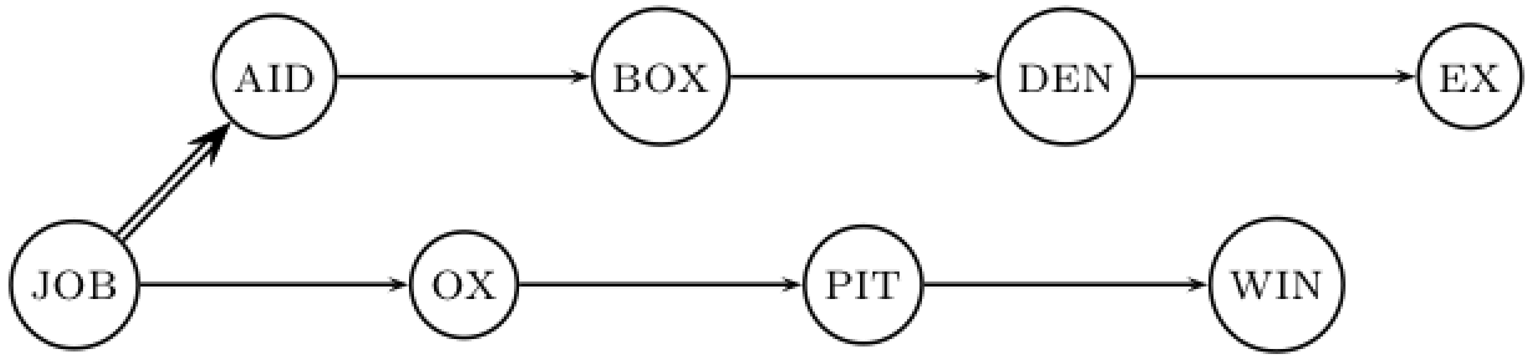
Space for dictionary as a string with blocking

- Example block size k = 4
- Where we used 4 $\times$ 3 bytes for term pointers without blocking . . .
- . . .we now use 3 bytes for one pointer plus 4 bytes for indicating the length of each term.
- We save 12 − (3 + 4) = 5 bytes per block.
- Total savings: 400,000/4 $*$ 5 = 0.5 MB
- This reduces the size of the dictionary from 7.6 MB to 7.1
- MB.

Lookup of a term without blocking



Lookup of a term with blocking: (slightly) slower



Front coding

One block in blocked compression (*k* = 4) . . .

8 a u t o m a t a **8** a u t o m a t e **9** a u t o m a t i c **10** a u t o m a t i o n

$\Downarrow$

. . . further compressed with front coding.

8 a u t o m a t $*$ a **1** $\diamond$ e **2** $\diamond$ i c **3** $\diamond$ i o n

Dictionary compression for Reuters: Summary

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

Outline

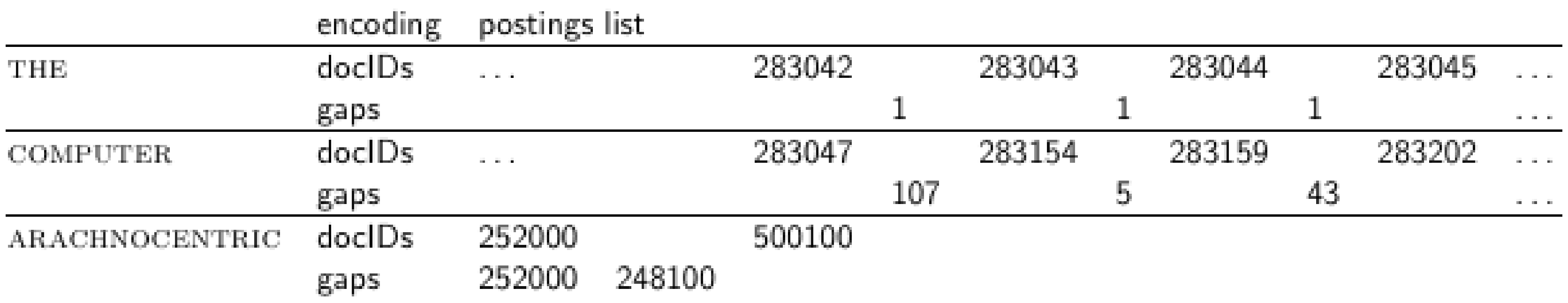
Postings compression

- 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 $\log_2 800{,}000 \approx 19.6 < 20$ bits per docID.
- Our goal: use a lot less than 20 bits per docID.

Key idea: Store gaps instead of docIDs

- Each postings list is ordered in increasing order of docID.
- Example postings list: COMPUTER: 283154, 283159, 283202, . . .
- It suffices to store gaps: 283159-283154=5, 283202-283154=43
- Example postings list using gaps : COMPUTER: 283154, 5, 43, . . .
- Gaps for frequent terms are small.
- Thus: We can encode small gaps with fewer than 20 bits.

Gap encoding



	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 and other rare terms, we will use about 20 bits per gap (= posting).
  - For THE and other very frequent terms, we will use only a few bits per gap (= posting).
- In order to implement this, we need to devise some form of variable length encoding.
- Variable length encoding uses few bits for small gaps and many bits for large gaps.

Variable byte (VB) code

- Used by many commercial/research systems
- Dedicate 1 bit (high bit) to be a continuation bit *c*.
- If the gap *G* fits within 7 bits, binary-encode it in the 7 available bits and set *c* = 1.
- Else: encode lower-order 7 bits and then use one or more 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 of the other bytes to 0 (*c* = 0).

VB code examples

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

VB code encoding algorithm

```
VBEncodeNumber($n$)
bytes $\leftarrow \langle\rangle$
while true
do Prepend(bytes, $n$ mod 128)
    if $n < 128$
      then Break
    $n \leftarrow n$ div 128
bytes[Length(bytes)] $+= 128$
return bytes
```

```
VBEncode(numbers)
bytestream $\leftarrow \langle\rangle$
for each $n \in$ numbers
do bytes $\leftarrow$ VBEncodeNumber($n$)
    bytestream $\leftarrow$ Extend(bytestream, bytes)
return bytestream
```

VB code decoding algorithm

```
VBDecode(bytestream)
numbers ← ⟨⟩
$n \leftarrow 0$
for $i \leftarrow 1$ to Length(bytestream)
do if $bytestream[i] < 128$
        then $n \leftarrow 128 \times n + bytestream[i]$
        else $n \leftarrow 128 \times n + (bytestream[i] - 128)$
             Append(numbers, n)
             $n \leftarrow 0$
return numbers
```

Other variable codes

- Instead of bytes, we can also use a different “unit of alignment”: 32 bits (words), 16 bits, 4 bits (nibbles) etc
- Variable byte alignment wastes space if you have many small gaps – nibbles do better on those.
- Recent work on word-aligned codes that efficiently “pack” a variable number of gaps into one word – see resources at the end

Gamma codes for gap encoding

- You can get even more compression with another type of variable length encoding: bitlevel code.
- Gamma code is the best known of these.
- First, we need unary code to be able to introduce gamma code.
- 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 70 is:

11111111111111111111111111111111111111111111111111111111111111111111110

Gamma code

- Represent a gap G as a pair of length and offset.
- Offset is the gap in binary, with the leading bit chopped off.
- For example 13 → 1101 → 101 = offset
- Length is the length of offset.
- For 13 (offset 101), this is 3.
- Encode length in unary code: 1110.
- Gamma code of 13 is the concatenation of length and offset: 1110101.

Gamma code examples

number	unary code	length	offset	γ code
0	0			
1	10	0		0
2	110	10	0	10,0
3	1110	10	1	10,1
4	11110	110	00	110,00
9	1111111110	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

- Compute the variable byte code of 130
- Compute the gamma code of 130

130 is (2 to the power 7) plus 2
130 = 0000000010000010
130 = 1111110000010

Length of gamma code

- The length of offset is $\lfloor \log_2 G \rfloor$ bits.
- The length of length is $\lfloor \log_2 G \rfloor + 1$ bits,
- So the length of the entire code is $2 \times \lfloor \log_2 G \rfloor + 1$ bits.
- Υ codes are always of odd length.
- Gamma codes are within a factor of 2 of the optimal encoding length $\log_2 G$.
  - (assuming the frequency of a gap G is proportional to $\log_2 G$ – not really true)

Gamma code: Properties

- Gamma code is prefix-free: a valid code word is not a prefix of any other valid code.
- Encoding is optimal within a factor of 3
- This result is independent of the distribution of gaps!
- We can use gamma codes for any distribution. Gamma code is universal.

Compression of Reuters

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)	3600.0
collection (text)	960.0
T/D 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, encoded	101.0

Term-document incidence matrix

	Anthony and Cleopatra	Julius Caesar	The Tempest	Hamlet	Othello	Macbeth	. . .
ANTHONY	1	1	0	0	0	1	
BRUTUS	1	1	0	1	0	0	
CAESAR	1	1	0	1	1	1	
CALPURNIA	0	1	0	0	0	0	
CLEOPATRA	1	0	0	0	0	0	
MERCY	1	0	1	1	1	1	
WORSER	1	0	1	1	1	0	
. . .							

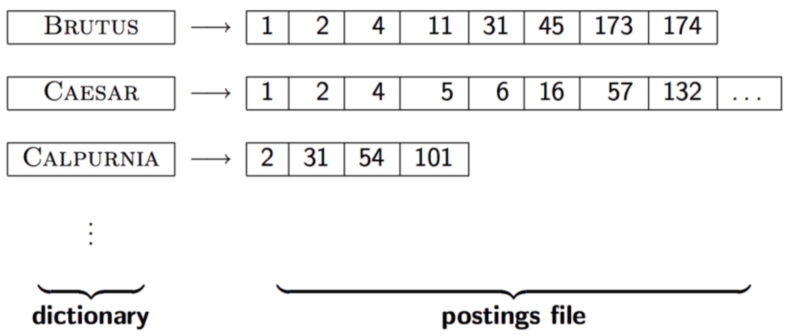
Entry is 1 if term occurs. Example: CALPURNIA occurs in *Julius Caesar*. Entry is 0 if term doesn't occur. Example: CALPURNIA doesn't occur in *The tempest*.

Summary

- We can now create an index for highly efficient Boolean retrieval that is very space efficient.
- Only 10-15% of the total size of the text in the collection.
- However, we've ignored positional and frequency information.
- For this reason, space savings are less in reality.

Take-away today

For each term t, we store a list of all documents that contain t.


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

- Motivation for compression in information retrieval systems
- How can we compress the dictionary component of the inverted index?
- How can we compress the postings component of the inverted index?
- Term statistics: how are terms distributed in document collections?

Entropy

1) Entropy = measure of randomness
2) Entropy = measure of compressibility

More random = Less compressible

Entropy enables one to compute the compressibility of data without actually needed to compress the date first!

For instance, entropy can predict a 50% file compressibility. This means that if compression is carried out, the original file should shrink by 1/2

Huffman encoding example

We compute the Huffman code and measure the amount of compression of the file.

This is compared to the “entropy”, a measure of file compressibility obtained from the file (without the need to actually compress)

!

Probabilities

6-sided fair dice

pi = [Probability of outcome = i] = 1/6
Where i is any number from 1 to 6

6-sided biased dice

p6 = ¼, p1 = p2 = p3 = p4 = p5 = 3/20

Sum of probabilities of all possible outcomes is 1

p1 + p 2 + p3 + p4 + p5 + p6 = 1

Probabilities (general case)

For the general case, instead of 6 outcomes (dice) we can have n outcomes.

For instance, consider a file with 100 characters. Say the character a occurs 45 times in this file. What is the probability of encountering "a" if we pick a character in the file at random?

Answer: 45/100

Say there are l characters in the file.
Each character will have a probability pi.
Sum of the probabilities (p1 + …. + pn) = 100/100 (Explain). Hence again, we have sum 1

Entropy

Given a probability distribution, i.e. given a finite number of probabilities, p1, … , pn, where the sum of these is 1, then we define the entropy H of this distribution as:

H(p1,…,pn) = -p1log(p1) -p2log(p2) - … - pnlog(pn)

Exercise: compute the entropy for

a) the probability distribution of the fair dice

b) the probability distribution of the biased dice

Binary represenation

To encode n distinct numbers in binary notation
we need to use binary numbers of length log(n)
Note that from here on “log” will be the logarithm in base 2
since we are interested in binary compression only.

To encode 6 numbers, we need to use binary numbers of length log(3) (in fact, we need to take the nearest integer above this value, i.e. 8). Binary numbers of length 2 will not suffice (there are only 4 which is not suitable to encode 6 numbers). We keep matters as an approximation and talk about binary numbers of “length” log(6), even though this is not an integer value.

Binary number length to encode 8 numbers is log(8) = 3

Exercise: solution

a) Fair dice: p1 = p2 = … = p6 = 1/6
So H(p1,…,p6) = -1/6log(1/6) X 6
= - log(1/6)
= --log(6)
= log(6) = 2.59

Interpretation: Entropy measures the amount of randomness. In the case of a fair dice, the randomness is "maximum". All 6 outcomes are equally likely. This means that to represent the outcomes we will "roughly" need log(6) = 2.59 bits to represent them in binary form (the form compression will take).

Solution continued

b) The entropy for the biased dice is:

-1/4log(1/4) – 3/20log(3/20) x 5 =

1/4log(4) + 3/20log(20/3) x 5 =

¼ x 2 + 3/4 log(20/3) =

½ + 3/4log(20/3) =

0.5 + 2.74 =

3.24

Exercise continued

Try the same for an 8-sided dice
(dungeons and dragons dice) which is
a) Fair
b) Totally biased, with prob(8) =1
 and thus prob(1) = … = prob(7) = 0

Answers:
a) Entropy is log(8) = 3, we need 3 bits to represent the 8 outcomes (maximum randomness)
b) Entropy is 1log(1) = 0, we need a bit of length 0 to represent the outcome. Justify! (Note: bit of length 1 has 2 values. Bit of length 0 has ? Values).

Compression

Revisit previous example of 8-sided dice

Compression for outcomes of fair dice:

0% compression (we still need 8 values to encode)
Average entropy = 1 ("Total" randomness)
(outcome of entropy/total number of values) = 8/8 = 1

Compression for outcomes of biased dice:

100% compression (we only need 1 bit to encode)
Average entropy 0 ("No" randomness)
(outcome of entropy/total number of values) = 0/8 = 0

Exercise: Huffman code

Consider a file with the following properties:

Characters in file:	a,b,c,d,e and f
Number of characters:	100
Frequencies of characters:	freq(a) = 45, freq(b) = 13, freq(c) = 12, freq(d) = 16, freq(e) = 9, freq(f) = 5.

Exercise continued

a) Compute the Huffman encoding
b) Compute the cost of the encoding
c) Compute the average length of the encoding
d) Express the probability of encountering a character in the file (do it for each character)
e) Compute the Entropy
f) Compare the Entropy to the compression percentage

What is your conclusion?

Solution

We assume familiarity with the Huffman code Algorithm.

Answer:
a) (prefix) codes for characters:
a: 0, b: 101, c:100, d: 111, e: 1101, f: 1100

b) Cost of encoding = number of bits in encoding =
45 x 1 + 13 x 3 + 12 x 3 + 16 x 3 + 9 x 4 + 5 x 4 = 224 bits

c) 224/100 = 2.24 average encoding length

d) Prob(char = a) = 45/100, …, Prob(char = f) = 5/100
Check: sum of probabilities = 100/100 = 1

Solution (continued)

e) Entropy =

H(45/100, 13/100, 12/100, 16/100, 9/100, 5/100) =

– 45/100log(45/100) – 13/100log(13/100)
– 12/100log(12/100) – 16/100log(16/100)
– 9/100log(9/100) – 5/100log(5/100) =) = 2.23

f) Conclusion: Entropy is an excellent prediction of average binary encoding length (some minor round-off errors). It predicted the average code length to be 2.23, very close to 2.24. It also predicts total size of compressed file: 2.23 x 100,000 = 223,000 which is very close to actual compressed size: 224,000