مبانی بازیابی اطلاعات و جستجوی وب

Index Compression-۵

Original slides by: Hinrich Schütze and Christina Lioma

Outline

1. Compression
2. Term statistics
3. Dictionary compression
4. Postings compression

Why compression in information retrieval?

- Space for **dictionary**
  - Main motivation for dictionary compression: make it small enough to keep in **main memory**
- Space 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**

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

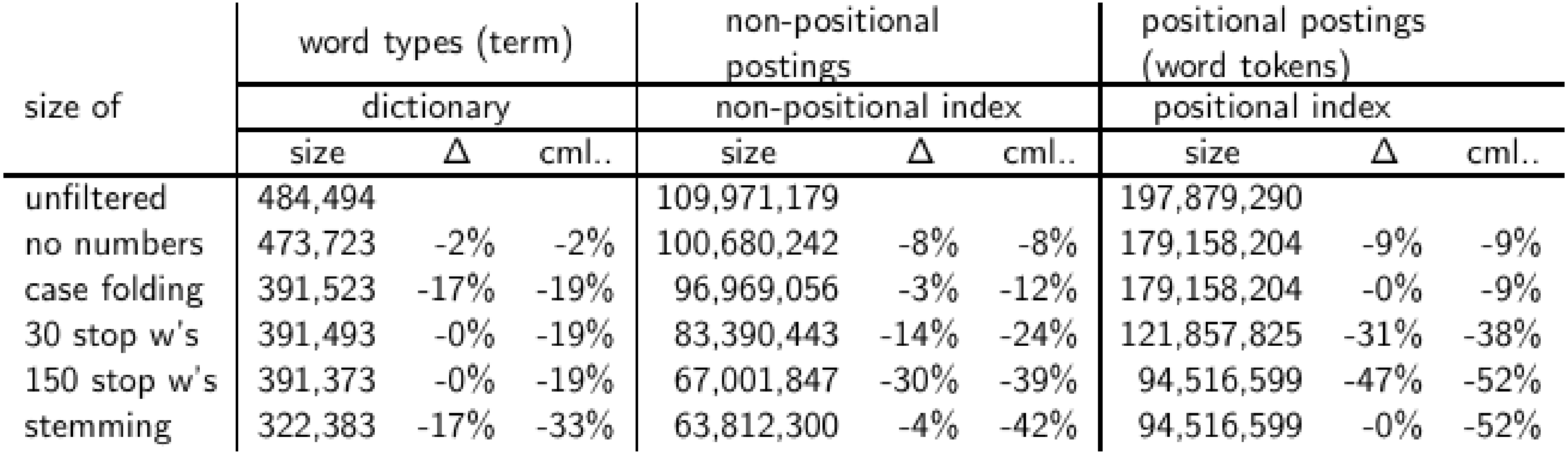
Outline

1. Compression
2. Term statistics
3. Dictionary compression
4. Postings compression

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



	word types (term)			non-positional postings			positional postings (word tokens)		
size of	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?

- The vocabulary will keep growing with collection size.
- Heaps’ law (enpirical): $M = kT^b$
- 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.

Empirical fit for Reuters

- For these data, $\log_{10} M = 0.49 * \log_{10} T + 1.64$
- Thus, $M = 10^{1.64} T^{0.49}$ and $k = 10^{1.64} \approx 44$ and $b = 0.49$.
- 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.

Zipf's law

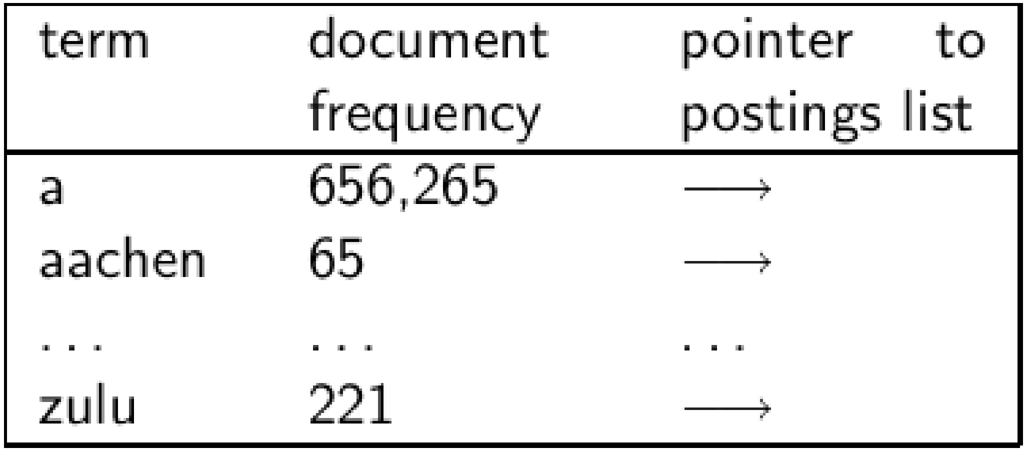
- We also want to know how many frequent vs. infrequent terms we should expect in a collection.
- In natural language, there are a few very frequent terms and very many very rare terms.
- Zipf's law: The i^{th} most frequent term has frequency cf_i proportional to $1/i$.
- cf_i is collection frequency: the number of occurrences of the term t_i in the collection.

  $\text{cf}_i \propto \frac{1}{i}$

Outline

1. Compression
2. Term statistics
3. Dictionary compression
4. Postings compression

Recall: Dictionary as array of fixed-width entries



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

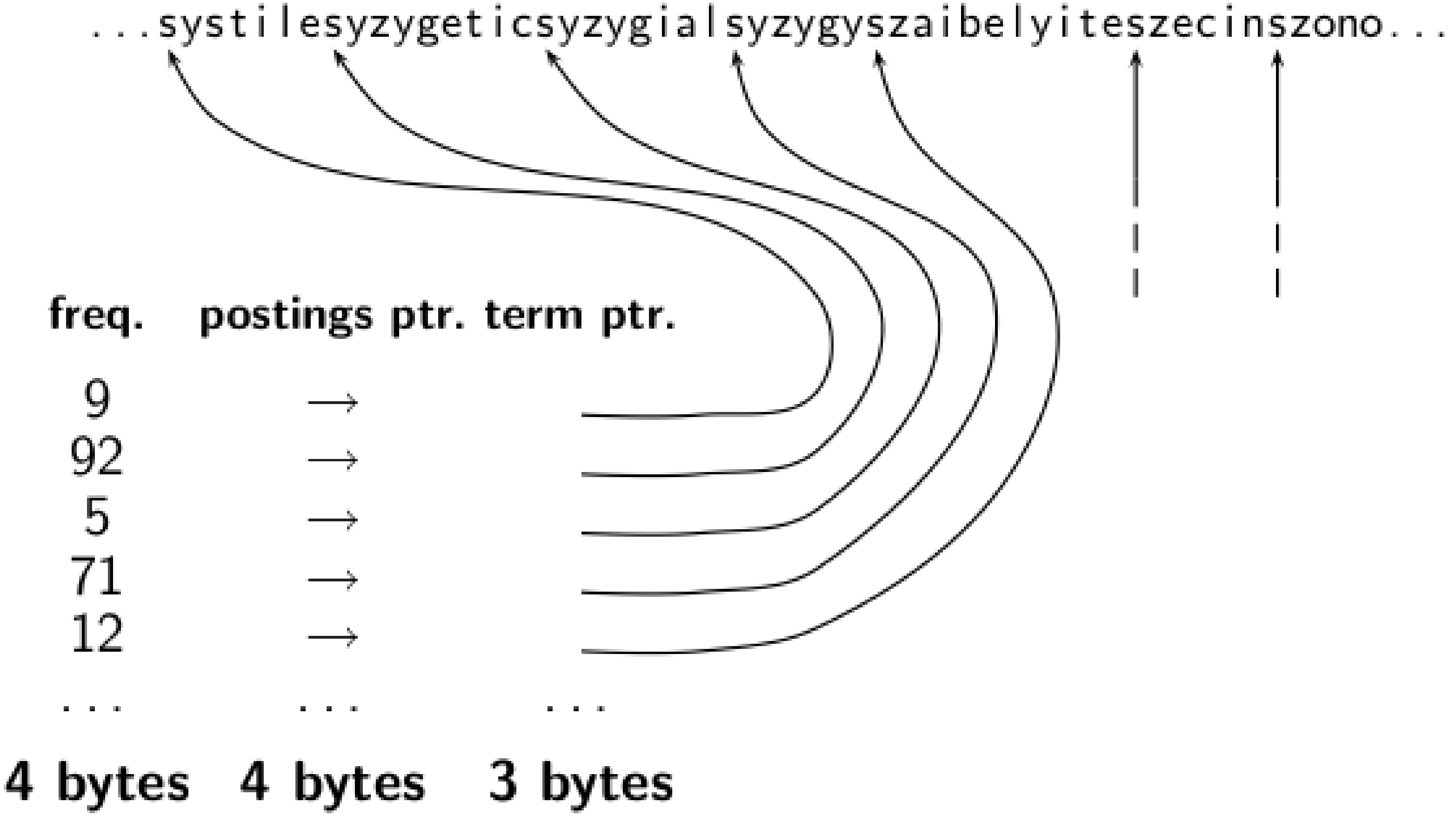
Space needed: 20 bytes 4 bytes 4 bytes

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
- Average length of a term in English: 8 characters
- How can we use on average 8 characters per term?

Dictionary as a string



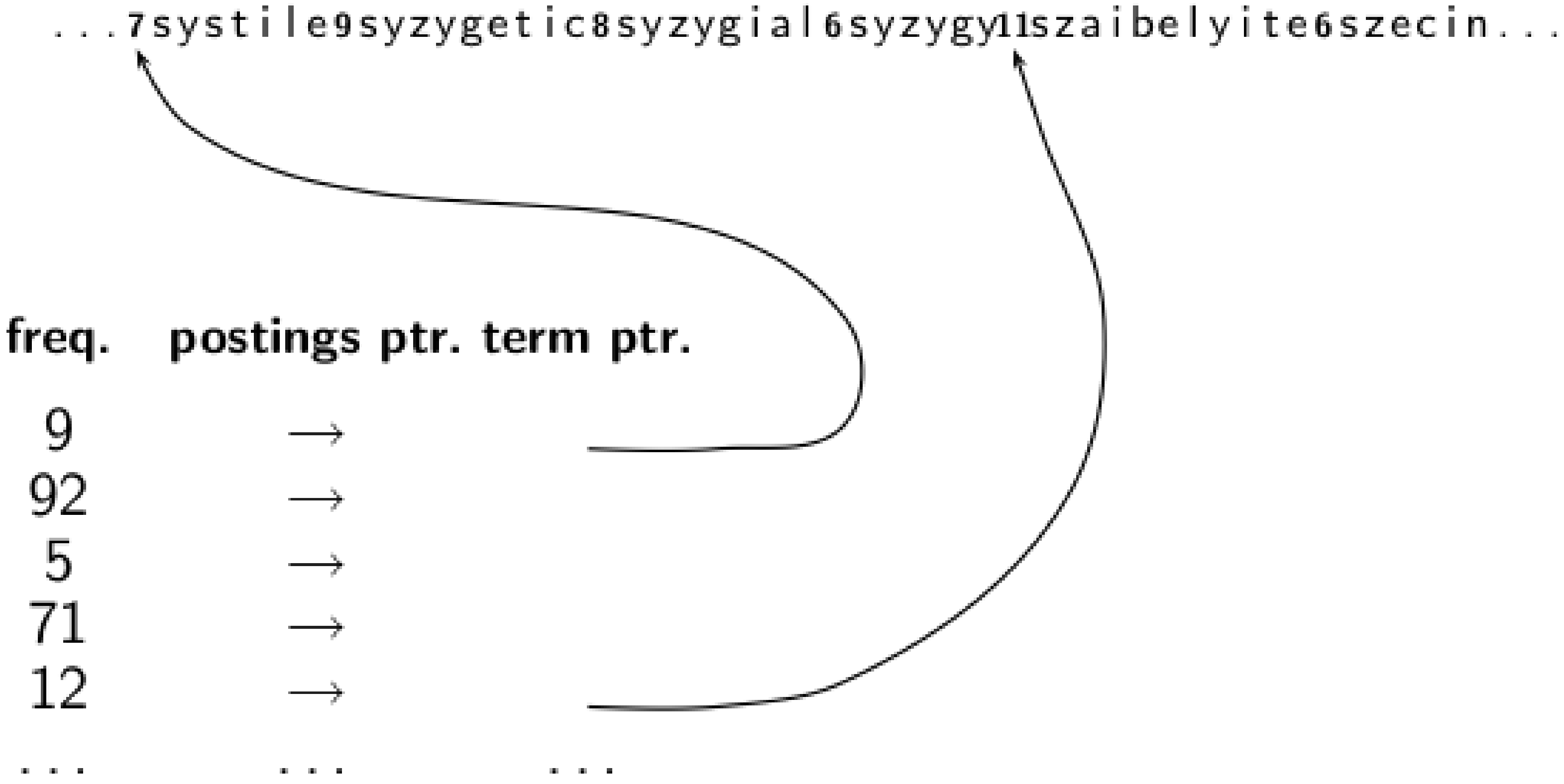
. . . systilesyzygeticsyzygialsyzygyszaibelyiteszecinszono . . .

freq.	postings ptr.	term ptr.
9	→	
92	→	
5	→	
71	→	
12	→	
. . .	. . .	. . .
4 bytes	**4 bytes**	**3 bytes**

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 × (4 +4 +3 + 8) = 7.6MB (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 × 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.

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

Outline

1. Compression
2. Term statistics
3. Dictionary compression
4. Postings compression

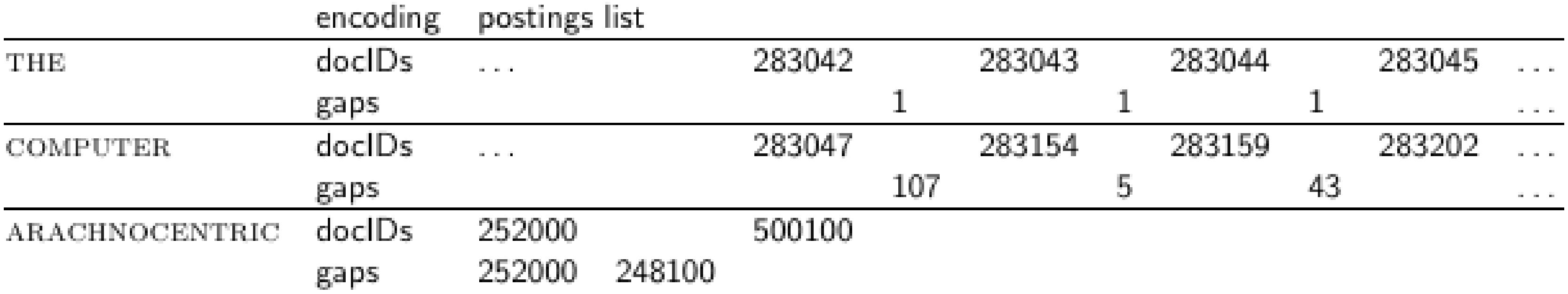
Postings compression

- The postings file is much larger than the dictionary, factor of at least 10.
- 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, 283202-283159=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 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 byte (VB) code

- 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

Optional: Gamma codes for gap encoding

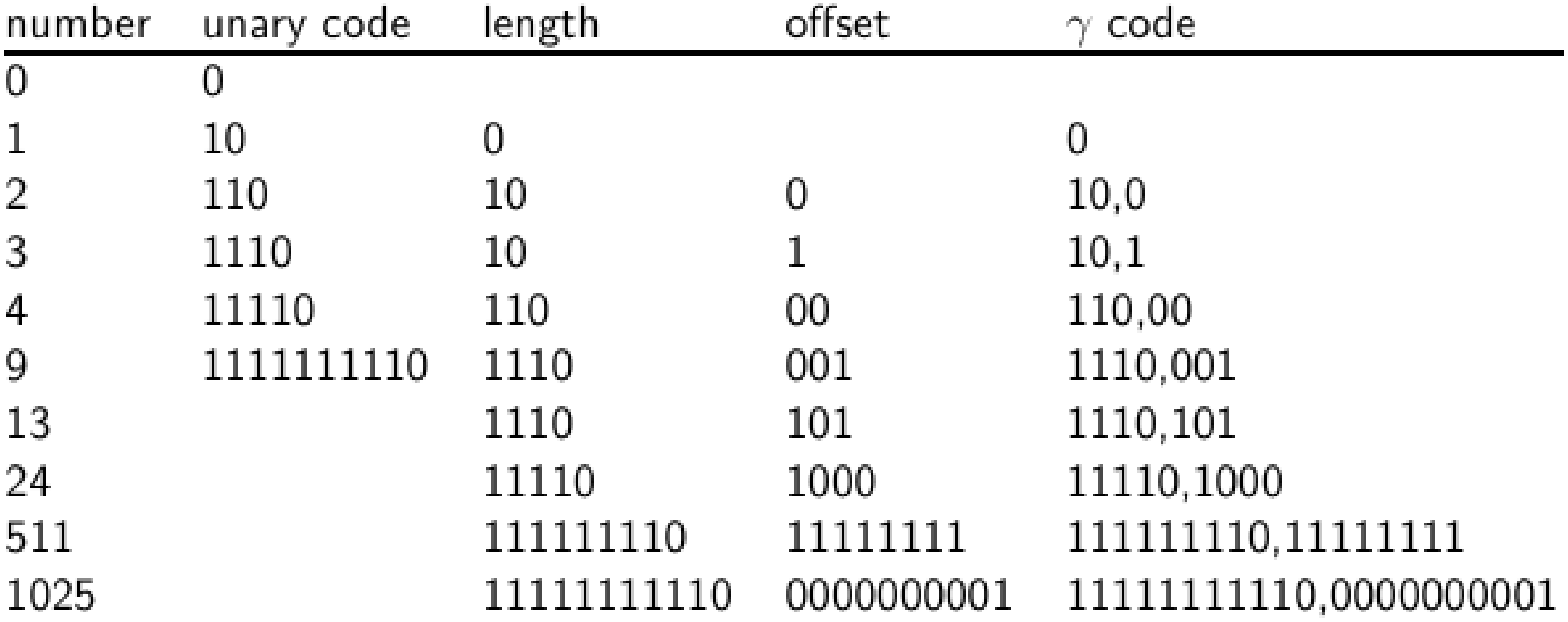
- You can get even more compression with another type of variable length encoding: bitlevel code.
- Gamma code is the best known of these.
- First, we need unary code to be able to introduce gamma code.
- Unary code
  - Represent *n* as *n* 1s with a final 0.
  - Unary code for 3 is 1110
  - Unary code for 40 is 11111111111111111111111111111111111111110
  - Unary code for 70 is:

11111111111111111111111111111111111111111111111111111111111111111111110

Optional: 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.

Optional: 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

Optional: 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 x $\lfloor \log_2 G \rfloor + 1$ 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 $\log_2 \quad G$ – not really true)
- Gamma code is parameter-free.

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

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.

منابع

- فصل پنجم کتاب An introduction to information retrieval