Modern Information Retrieval

Index compression[1]

Hamid Beigy

Sharif university of technology

October 13, 2023

[1] Some slides have been adapted from slides of Manning, Yannakoudakis, and Schütze.

Table of contents

Introduction

1. Dictionary and inverted index: core of IR systems
2. Techniques can be used to compress these data structures, with two objectives:
   - reducing the disk space needed
   - reducing the time processing, by using a cache (keeping the postings of the most frequently used terms into main memory)
3. Decompression can be faster than reading from disk

Table of contents

Characterization of an index

Considering the Reuters-RCV1 collection

size of	dictionary			non-positional index			positional index		
	size	Δ	cum.	size	Δ	cum.	size	Δ	cum.
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 words	391,493	-0%	-19%	83,390,443	-14%	-24%	121,857,825	-31%	-38%
150 stop words	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%

1. The vocabulary grows with the corpus size
2. Empirical law determining the number of term types in a collection of size M (Heap's law)

$$M = kT^b$$

where T is the number of tokens, and k and b are two parameters defined as follows:

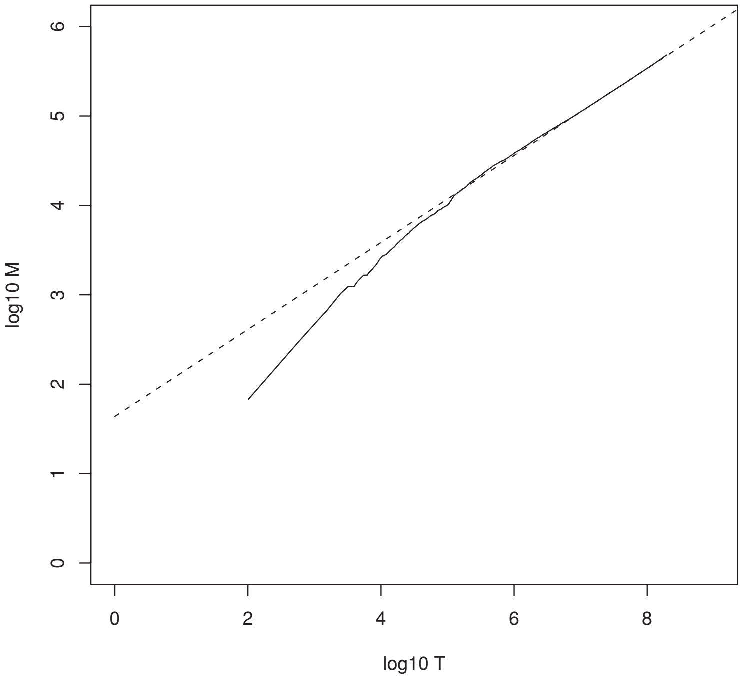
$$b \approx 0.5$$
$$30 \leq k \leq 100$$

(k is the growth-rate)

3. On the REUTERS corpus for the first $1,000,020$ tokens (taking $k = 44$ and $b = 0.49$):

$$M = 44 \times 1,000,020^{0.5} = 38,323$$

Statistical properties of terms (Heap's law)



Modeling the distribution of terms (Zipf's law)

1. Now we have characterized the growth of the vocabulary in collections.
2. We want understand how terms are distributed across documents.
3. We want know how many frequent vs. infrequent terms we should expect in a collection.
4. In natural language, there are a few very frequent terms and very many very rare terms.
5. **Zipf's law:** The i^{th} most frequent term has frequency cf_i proportional to $\frac{1}{i}$.

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

6. cf_i is collection frequency: the number of occurrences of the term t_i in the collection.
7. So if the most frequent term (*the*) occurs cf_1 times, then

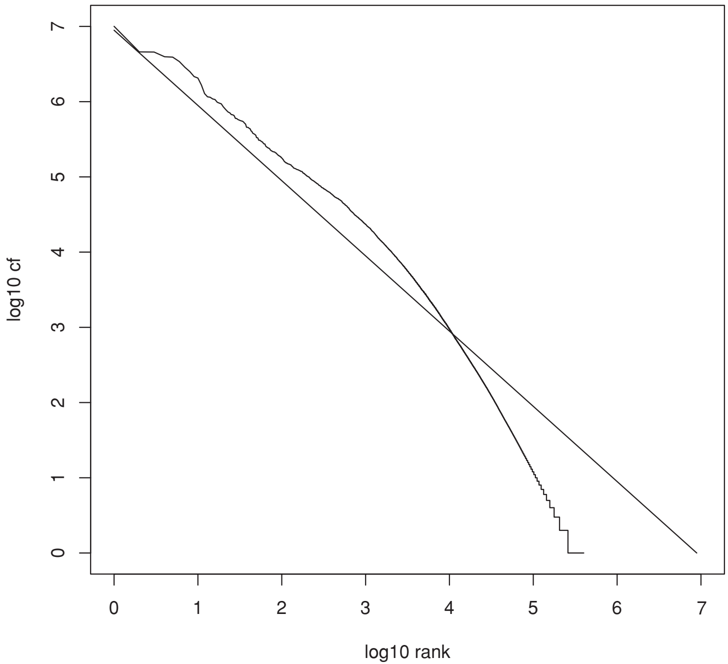
$$\begin{aligned}
\mathrm{cf}_2 &= \frac{1}{2}\mathrm{cf}_1 \\
\mathrm{cf}_3 &= \frac{1}{3}\mathrm{cf}_1 \\
&\vdots \\
\mathrm{cf}_k &= \frac{1}{k}\mathrm{cf}_1
\end{aligned}$$

8. Equivalently, we can write Zipf's law as

$$\mathrm{cf}_i = c i^k$$

$$\log \mathrm{cf}_i = \log c + k \log i \quad \text{for} \quad k = -1$$

Modeling the distribution of terms (Zipf's law)



Dictionary compression

Dictionary compression

1. The dictionary is small compared to the postings file.
2. But we want to keep it in memory.
3. We compress the dictionary because of
    - Reduce the response time of an IR system
    - We want design the search system for systems with limited hardware such as cell phones, onboard computers.
    - Fast startup time
    - Sharing resurces with other applications.
4. So compressing the dictionary is important.

Index format with fixed-width entries

term	document frequency	pointer to postings list	postings list
a	656,265	$\longrightarrow$	...
aachen	65	$\longrightarrow$	...
...	...	...	...
zulu	221	$\longrightarrow$	...
40	4	4	space needed

1. Total space for using Unicode and fixed-width entries (term-length=20):

$$M \times (2 \times 20 + 4 + 4) = 400{,}000 \times 48 = 19.2 \text{ MB}$$

2. Without using Unicode:

$$M \times (20 + 4 + 4) = 400{,}000 \times 28 = 11.2 \text{ MB}$$

3. Remarks
   - The average length of a word type for REUTERS is 7.5 bytes
   - With fixed-length entries, a one-letter term is stored using 20 bytes!
   - Some very long words (such as hydrochlorofluorocarbons) cannot be handled.
   - How can we extend the dictionary representation to save bytes and allow for long words?

Compressing the dictionary

Dictionary as a string

. . . s y s t i l e s y z y g e t i c s y z y g i a l s y z y g y s z a i b e l y

freq.	**postings ptr.**	**term ptr.**
9	$\rightarrow$	
92	$\rightarrow$	
5	$\rightarrow$	
71	$\rightarrow$	
12	$\rightarrow$	
. . .	. . .	. . .
4 bytes	**4 bytes**	**3 bytes**

1. 3 bytes per pointer into string (need $\log_2 400000 \approx 22$ bits to resolve 400,000 positions)
2. 8 chars (on average) for term in string
3. Using Unicode: $400{,}000 \times (4 + 4 + 3 + 2 \times 8) = 10.8$MB (compared to 19.2 MB for fixed-width)
4. Without using Unicode: $400{,}000 \times (4 + 4 + 3 + 8) = 7.6$ MB (compared to 11.2 MB for fixed-width)

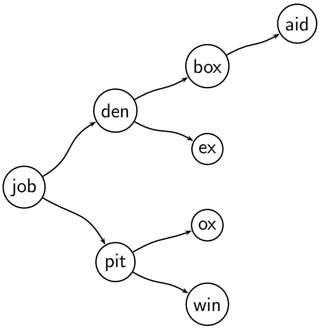
Block storage

. . . **7** s y s t i l e **9** s y z y g e t i c **8** s y z y g i a l **6** s y z y g y **11** s z a i b e l y i t

freq.	**postings ptr.**	**term ptr.**
9	$\rightarrow$	
92	$\rightarrow$	
5	$\rightarrow$	
71	$\rightarrow$	
12	$\rightarrow$	

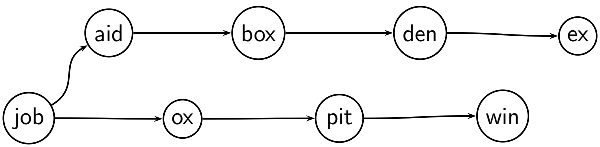
1. Let us consider blocks of size k
2. We remove $k-1$ pointers, but add k bytes for term length
3. Example:
   $k = 4$, $(k-1) \times 3$ bytes saved (pointers), and 4 bytes added (term length) $\rightarrow$ 5 bytes saved
4. Space saved:
   $400{,}000 \times (\frac{1}{4}) \times 5 = 0.5$ MB (dictionary reduced to 10.3 MB and for non-Unicode 7.1MB)
5. Why not taking $k > 4$?

1. Uncompressed dictionary



Average search cost: $(0+1+2+3+2+1+2+2)/8 \approx 1.6$ steps

2. Compressed dictionary with blocking



Average search cost: $(0+1+2+3+4+1+2+3)/8 \approx 2$ steps

Front coding

1. Many words have the same prefix. We can write common prefix once.
2. One block in blocked compression ($k = 4$)

   8automata8automate9automatic10automation

3. Compressed with front coding.

   8automat$*$a1$\diamond$e2$\diamond$ic3$\diamond$ion

4. End of prefix marked by $*$
5. Deletion of prefix marked by $\diamond$

Dictionary compression for Reuters

representation	size (unicode)	size (non-unicode)
dictionary, fixed-width	19.2MB	11.2MB
dictionary as a string	10.8MB	7.6MB
$\sim$, with blocking, $k = 4$	10.3MB	7.1MB
$\sim$, with blocking & front coding	7.9MB	5.9MB

Compressing the posting lists

Compressing the posting lists

1. The REUTERS collection has
   - about 800 000 documents,
   - each having 200 tokens
2. Since tokens are encoded using 6 bytes, the collection's size is 960 MB
3. A document identifier must cover all the collection, *i.e.* must be $log_2 800,000 \approx 20$ bits
4. If the collection includes about $100,000,000$ postings, the size of the posting lists is $100,000,000 \times 20/8 = 250MB$
5. How to compress these postings ?
6. Idea: most frequent terms occur close to each other.
7. **We encode the gaps between occurrences of a given term**

	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								

Furthermore, small gaps are represented with shorter codes than big gaps.

Two techniques

- Variable-length byte-codes (Byte-level)
- γ-codes (Bit-level)

Compressing the posting lists

Using variable-length byte-codes

Using variable-length byte-codes

1. Variable-length byte encoding uses an integral number of bytes to encode a gap
2. First bit := *continuation byte*
3. Last 7 bits := *part of the gap*
4. The first bit is set to 1 for the last byte of the encoded gap, 0 otherwise
5. Example: a gap of size 5 is encoded as 10000101

Example

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

What is the code for a gap of size 1283?

6. The posting lists for the REUTERS collection are compressed to 116 MB with this technique (original size: 250 MB)
7. The idea of representing gaps with variable integral number of bytes can be applied with units that differ from 8 bits
8. Larger units can be processed (decompression) quicker than small ones, but are less effective in terms of compression rate

Compressing the posting lists

Using γ-codes

Using γ-codes

1. Idea: representing numbers with a *variable bit* code
2. **Unary code:** the number n is encoded as: $\overbrace{11\ldots}^{\text{n times}}0$
   (not efficient)
3. **γ-code:** variable encoding done by splitting the representation of a gap as follows:

length	offset

   - *offset* is the binary encoding of the gap (without the leading 1)
   - *length* is the unary code of the offset size

Unary and γ-codes

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

Example

1. Given the following γ-coded gaps:

   1110001110101011111101101111011

2. Decode these, extract the gaps, and recompute the posting list
3. γ-decoding :
   - first reads the length (terminated by 0),
   - then uses this length to extract the offset,
   - and eventually prepends the missing 1

   1110001 - 11010 - 101 - 11111011011 - 11011

Compression of Reuters: Summary

representation	size in MB Unicode	size in MB non-unicode
dictionary, fixed-width	19.2	11.2
dictionary, term pointers into string	10.8	7.6
$\sim$, with blocking, $k = 4$	10.3	7.1
$\sim$, with blocking & front coding	7.9	5.3
collection (text, xml markup etc)	3600.0	3600.0
collection (text)	960.0	960.0
term incidence matrix	40,000.0	40,000.0
postings, uncompressed (32-bit words)	400.0	400.0
postings, uncompressed (20 bits)	250.0	250.0
postings, variable byte encoded	116.0	116.0
postings, γ encoded	101.0	101.0

Conclusion

Conclusion

1. γ-codes achieve better compression ratios (about 15 % better than variable bytes encoding), **but** are more complex (expensive) to decode
2. This cost applies on query processing $\rightarrow$ trade-off to find
3. The objectives announced are met by both techniques, recall:
   - reducing the disk space needed
   - reducing the time processing, by using a *cache*
4. The techniques we have seen are *lossless compression* (no information is lost)
5. *Lossy compression* can be useful, *e.g.* storing only the most relevant postings (more on this in the ranking lecture)

References

Reading

1. Chapters 5 of Information Retrieval Book[2]

[2] Christopher D. Manning, Prabhakar Raghavan, and Hinrich Schütze (2008). **Introduction to Information Retrieval.** New York, NY, USA: Cambridge University Press.

References

Manning, Christopher D., Prabhakar Raghavan, and Hinrich Schütze (2008). **Introduction to Information Retrieval.** New York, NY, USA: Cambridge University Press.

Questions?