# Information Retrieval

## Index compression

Hamid Beigy

Sharif university of technology

October 19, 2018

# Introduction

1. Dictionary and inverted index: core of IR systems
2. Techniques can be used to compress these data structures, with two objectives:
   - educing 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

# Table of contents

# Characterization of an index

1. Considering the Reuters-RCV1 collection

| | word types | | | non-positional postings | | | positional postings (word tokens) | | |
|---|---|---|---|---|---|---|---|---|---|
| size of | dictionary | | | non-positional index | | | positional index | | |
| | size | Δ | cumul. | size | Δ | cumul. | size | Δ | cumu |
| 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% |

# Statistical properties of terms

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 $b$
2 parameters defined as follows:

$$b \approx 0.5 \text{ and } 30 \leq k \leq 100$$

($k$ is the growth-rate)
3. On the REUTERS corpus fo the first $1,000,020$ tokens (taking $k = 44$ and $b = 0.49$):

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

# Index format with fixed-width entries

| | term | tot. freq. | pointer to postings list | postings list |
|---|---|---|---|---|
| | a | 656,265 | $\longrightarrow$ | ... |
| | aachen | 65 | $\longrightarrow$ | ... |
| | ... | ... | ... | ... |
| | zulu | 221 | $\longrightarrow$ | ... |
| space needed: | 40 bytes | 4 bytes | 4 bytes | |

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

why 40 bytes per term ? (unicode + max. length of a term)
Without using unicode: $M \times (20 + 4 + 4) = 400{,}000 \times 28 = 11.2$ MB

# Remarks

1. The average length of a word type for REUTERS is 7.5 bytes
2. With fixed-length entries, a one-letter term is stored using 40 bytes!
3. Some very long words (such as hydrochlorofluorocarbons) cannot be handle
4. How can we extend the dictionary representation to save bytes and allow for long words ?

# Table of contents

# 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 | → | |
| 92 | → | |
| 5 | → | |
| 71 | → | |
| 12 | → | |
| . . . | . . . | . . . |
| **4 bytes** | **4 bytes** | **3 bytes** |

# Space use for dictionary-as-a-string

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

# Block storage

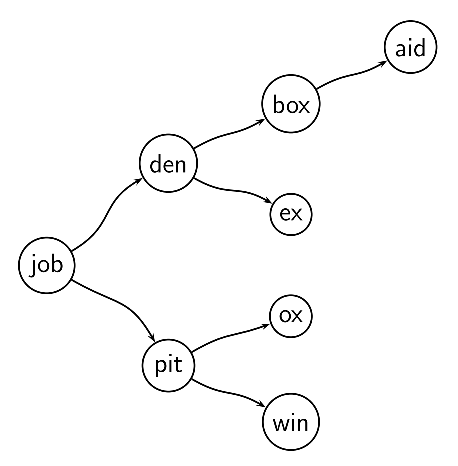
. . . **7**systile**9**syzygetic**8**syzygial**6**syzygy**11**szaibelyit

| **freq.** | **postings ptr.** | **term ptr.** |
|---|---|---|
| 9 | → | |
| 92 | → | |
| 5 | → | |
| 71 | → | |
| 12 | → | |

# Space use for block-storage

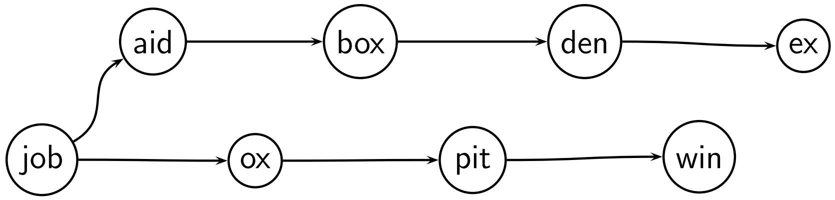
1. Let us consider blocks of size $k$
2. We remove $k-1$ pointers, but add $k$ bytes for term length
3. Example: $k=4$, $(k-1)\times 3$ bytes saved (pointers), and 4 bytes added (term length) $\rightarrow$ 5 bytes saved
4. Space saved: $400{,}000 \times (\frac{1}{4}) \times 5 = 0.5$ MB (dictionary reduced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

# Search without blocking




Average search cost: $(4 + 3 + 2 + 3 + 1 + 3 + 2 + 3)/8 \approx 2.6$ steps

# Search with blocking




Average search cost: $(2+3+4+5+1+2+3+4)/8 \approx 3$ steps

# 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

- End of prefix marked by $*$
- Deletion of prefix marked by $\diamond$

# Dictionary compression for Reuters

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

# Table of contents

# Compressing the posting lists

1. Recall: 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 $log_2 800000 \approx 20$ bits long
4. If the collection includes about 100 000 000 postings, the size of the posting lists is $100000000 \times 20/8 = 250MB$
5. How to compress these postings ?
6. Idea: most frequent terms occur close to each other
   $\rightarrow$ we encode the gaps between occurences of a given term

# 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 | | | | | | | | |

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

# 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

# Variable-length byte code: 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?

# Variable-length byte code

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

# Using $\gamma$-codes

1. Idea: representing numbers with a *variable bit* code
2. Example: *unary code*

   the number $n$ is encoded as: $\overbrace{11\ldots}^{\text{n times}}\ 0$
   (not efficient)
3. $\gamma$-code, variable encoding done by splitting the representation of a gap as follows:

   | length | offset |
   |---|---|

4. *offset* is the binary encoding of the gap (without the leading 1)
5. *length* is the unary code of the offset size
6. Objective: having a representation that is as close as possible to the $log_2 G$ size (in terms of bits) for $G$ gaps

# Unary and $\gamma$-codes

| number | unary code | length | offset | $\gamma$ 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 |

$\gamma$-codes are always of odd length. More precisely, their length is $2 \times \lfloor log_2\ s \rfloor + 1$

# Example

1. Given the following $\gamma$-coded gaps:
   1110001110101011111101101111011
2. Decode these, extract the gaps, and recompute the posting list
3. $\gamma$-decoding :
   - first reads the length (terminated by 0),
   - then uses this length to extract the offset,
   - and eventually prepends the missing 1

# 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, $\gamma$ encoded | 101.0 | 101.0 |

# Table of contents

# Conclusion

1. $\gamma$-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)

# Reading

Please read chapter 5 of Information Retrieval Book.