Information Retrieval

Weike Pan

The slides are **adapted from those provided by Prof. Hinrich Schütze** at University of Munich (http://www.cis.lmu.de/~hs/teach/14s/ir/).

Chapter 5 Index compression

- 5.1 Statistical properties of terms in information retrieval
- 5.2 Dictionary compression
- 5.3 Postings file compression
- 5.4 References and further reading

Outline

- 5.1 Statistical properties of terms in information retrieval
- 5.2 Dictionary compression
- 5.3 Postings file compression
- 5.4 References and further reading

5.1 Statistical properties of terms in information retrieval

- Reuters RCV1 statistics
  - documents: N=800,000
  - tokens per document: L=200
  - terms: M=400,000
  - bytes per token (including spaces/punctuations): 6
  - bytes per token (without spaces/ punctuations): 4.5
  - bytes per term: 7.5
  - non-positional postings: T=100,000,000

5.1 Statistical properties of terms in information retrieval

Heaps' law

- How many distinct words are there? Can we assume there is an upper bound? The vocabulary will keep growing with the collection size.

- **Heaps' law**: $M = kT^b$
  - M is the vocabulary size, T is the number of tokens in the collection.
  - Typical values for the parameters k and b are: $30 \le k \le 100$ and $b \approx 0.5$.
  - It is the simplest possible relationship between the collection size T and the vocabulary size M in the log-log space. It is an empirical law.

- **Question**: Can you verify the Heaps' law using the RCV1 data? (Plot the statistics in a $\log_{10} M$ - $\log_{10} T$ space)

5.1 Statistical properties of terms in information retrieval

Zipf's law

- How many frequent vs. infrequent terms we should expect in a collection?
- **Zipf's law**: cf_i is the collection frequency of the i-th most frequent term t_i, i.e., the number of occurrences of the term t_i in the collection.
  – cf_i is proportional to 1/i
- Question: Can you verify Zipf's law using the RCV1 data?

Outline

- 5.1 Statistical properties of terms in information retrieval
- 5.2 Dictionary compression
- 5.3 Postings file compression
- 5.4 References and further reading

5.2 Dictionary compression

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

5.2 Dictionary compression

Dictionary as an array of fixed-width entries (1/2)

	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

- Space for RCV1 data: (20+4+4)*400,000 = 11.2 MB

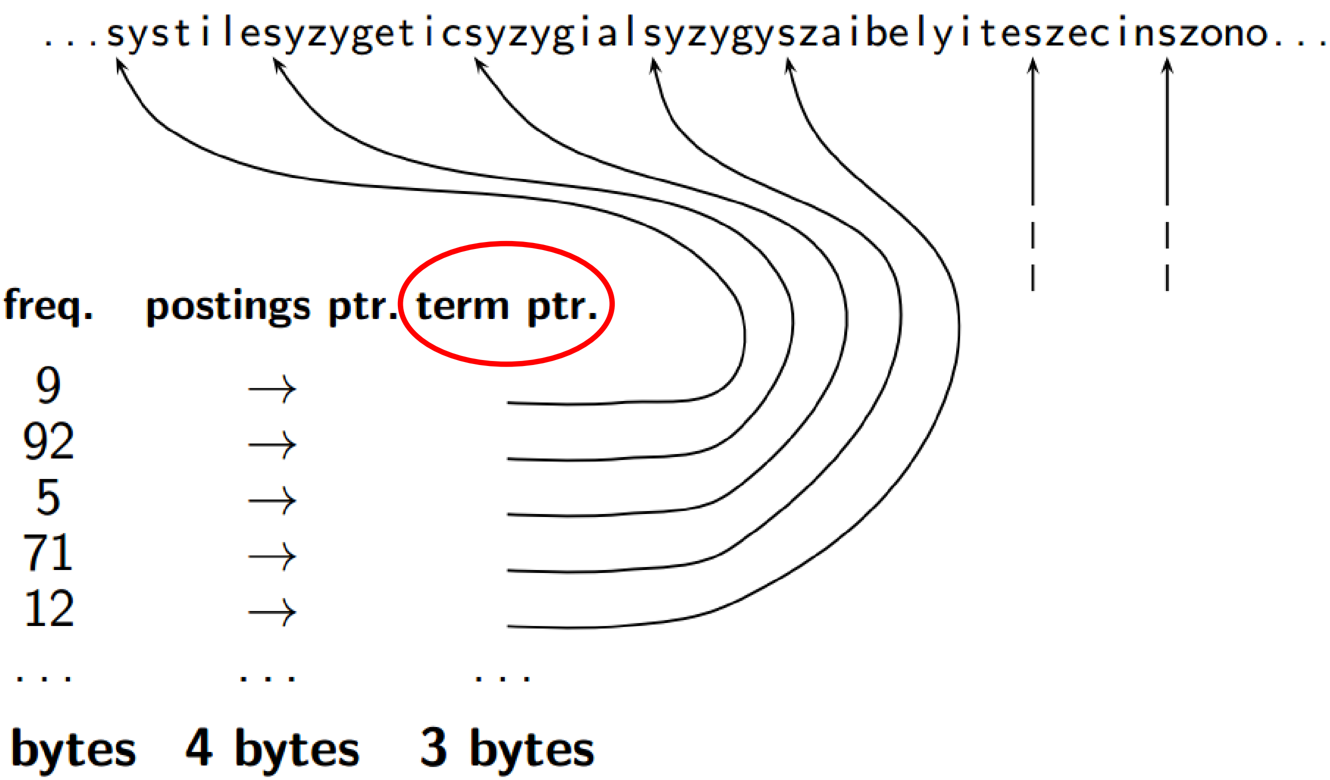
5.2 Dictionary compression

Dictionary as an array of fixed-width entries (2/2)

- Most of the bytes in the term column are wasted. We allot (分配) 20 bytes for terms of length 1.
- But, we still can not handle hydrochlorofluorocarbons and supercalifragilisticexpialidocious.
- Average length of a term in English: 8 characters (or a little bit less)
- How can we use "on average 8 characters per term"?

5.2 Dictionary compression

Dictionary as a string (1/2)



5.2 Dictionary compression

Dictionary as a string (2/2)

- 4 bytes per term for document frequency
- 4 bytes per term for pointer to postings list
- 8 bytes (on average) for term in string
- 3 bytes per pointer into string (needs log2(8 * 400000) = 21.6 < 24 bits to resolve 8 * 400000 positions)

- Space: 400000 * (4 + 4 + 8 + 3) = 7.6MB (compared to 11.2 MB for fixed-width array)

5.2 Dictionary compression

Dictionary as a string with blocking (1/2)

...7systile9syzygetic8syzygial6syzygy11szaibelyite6szecin...

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

5.2 Dictionary compression

Dictionary as a string with blocking (2/2)

- Example: block size k = 4
  - we use 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: 400000/4 * 5 = 0.5 MB

- This reduces the size of the dictionary from 7.6 MB to 7.1 MB.

5.2 Dictionary compression

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

5.2 Dictionary compression

Dictionary compression for the RCV1 data: Summary

data structure	size in MB
dictionary, fixed-width	11.2
dictionary, term pointers into string	7.6
$\sim$, with blocking, $k = 4$	7.1
$\sim$, with blocking & front coding	5.9

Outline

- 5.1 Statistical properties of terms in information retrieval
- 5.2 Dictionary compression
- 5.3 Postings file compression
- 5.4 References and further reading

5.3 Postings file compression

- The postings file (i.e., postings list) is much larger than the dictionary: factor of at least 10.

- A posting for our purpose is a docID.

- For the RCV1 data (800000 documents), we would use 32 bits per docID when using 4-byte integers.
- Alternatively, we can use log2 800000 ≈ 19.6 < 20 bits per docID.

- Our goal: use a lot less than 20 bits per docID.

5.3 Postings file compression

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.

5.3 Postings file compression

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								

5.3 Postings file compression

Variable length encoding

- Aim:
  - For ARACHNOCENTRIC and other rare terms, we will use about 20 bits per gap.
  - For THE and other very frequent terms, we will use only a few bits per gap.

- 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.

5.3 Postings file compression

Variable byte (VB) code

- Used by many commercial/research systems

- Dedicate one 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).

5.3 Postings file compression

VB code encoding algorithm

```
VBEncodeNumber(n)
  bytes ← ⟨⟩
  while true
  do Prepend(bytes, n mod 128)
     if n < 128
       then Break
     n ← n div 128
  bytes[Length(bytes)] += 128
  return bytes
```

```
VBEncode(numbers)
  bytestream ← ⟨⟩
  for each n ∈ numbers
  do bytes ← VBEncodeNumber(n)
     bytestream ← Extend(bytestream, bytes)
  return bytestream
```

- PREPEND: adds an element to the beginning of a list, e.g., PREPEND(< 1,2 >,3) = < 3,1,2 >.
- *bytes*[**LENGTH**(bytes)] += 128: **continuation bit (c=1)**
- EXTEND: extends a list, e.g., EXTEND(<1,2>, <3,4>) =<1,2,3,4>.

5.3 Postings file compression

VB code decoding algorithm

```
VBDecode(bytestream)
numbers ← ⟨⟩
n ← 0
for i ← 1 to Length(bytestream)
do if bytestream[i] < 128                              continuation bit (c=0)
      then n ← 128 × n + bytestream[i]
      else  n ← 128 × n + (bytestream[i] − 128)        continuation bit (c=1)
            Append(numbers, n)
            n ← 0
return numbers
```

5.3 Postings file compression

Example

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

5.3 Postings file compression

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.

5.3 Postings file compression

Gamma codes for gap encoding

- You can get even more compression with another type of variable length encoding: bit level 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

5.3 Postings file compression

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,

    e.g., 13->1101->101 = offset.
  - Length is the length of offset, e.g, for 13 (the offset is 101), this is 3. Encode length in unary code: 1110.
  - Gamma code of 13 is the concatenation of length and offset: 1110101.

5.3 Postings file compression

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

5.3 Postings file compression

Length of gamma code

- The length of offset is [log2 G] bits.
- The length of length is [log2 G] + 1 bits,
- So the length of the entire code is 2 * [log2 G] + 1 bits.

- Gamma codes are always of odd length.
- Gamma codes are within a factor of 2 of the optimal encoding length log2 G.

5.3 Postings file compression

Gamma code: Properties

- Gamma code (like variable byte code) is **prefix-free**: a valid code word is not a prefix of any other valid code.
- Gamma code is **parameter-free**.
- More theoretical analysis on the compression rate can be found in the textbook.

5.3 Postings file compression

Gamma codes: Alignment

- Machines have word boundaries, e.g., 8, 16, 32 bits.
- Compressing and manipulating at 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.

5.3 Postings file compression

Compression of the RCV1 data

data structure	size in MB
dictionary, fixed-width	11.2
dictionary, term pointers into string	7.6
$\sim$, with blocking, $k = 4$	7.1
$\sim$, 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, γ encoded	101.0

5.3 Postings file compression

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.

Summary

- 5.1 Statistical properties of terms in information retrieval
- 5.2 Dictionary compression
- 5.3 Postings file compression
- 5.4 References and further reading