

Information Retrieval and Organisation

Dell Zhang

Birkbeck, University of London

2015/16

Index Compression

Why Compression?

- ▶ Using less disk space
(saves money)
- ▶ Caching: keep more stuff in memory
(increases speed)
- ▶ Transferring data from disk to memory faster
(again, increases speed)
 - ▶ It would be faster to “read compressed data and decompress” than “read uncompressed data”
 - ▶ Premise: decompression algorithms are fast
 - ▶ This is true of the decompression algorithms that we will use

Why Compression in IR?

- ▶ For dictionary,
 - ▶ Main motivation: make it small enough to keep in main memory
- ▶ For postings file,
 - ▶ Main motivation: reduce disk space needed; decrease time needed to read from disk
 - ▶ Large search engines keep significant part of postings in memory
- ▶ We will devise various compression schemes

Lossy vs Lossless Compression

- ▶ Lossless compression: preserve all information
- ▶ Lossy compression: discard some information
 - ▶ Several of the preprocessing steps can be viewed as lossy compression
 - ▶ eliminating numbers
 - ▶ casefolding
 - ▶ stop words
 - ▶ stemming
 - ▶ What can we gain with lossy compression?

Preprocessing for Reuters

	(distinct) terms			non-positional postings			tokens (= number of position entries in postings)		
	number	$\Delta\%$	T%	number	$\Delta\%$	T%	number	$\Delta\%$	T%
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

- ▶ Lossy compression can be problematic, e.g. phrase queries
- ▶ We will focus on lossless compression for the remainder of this chapter

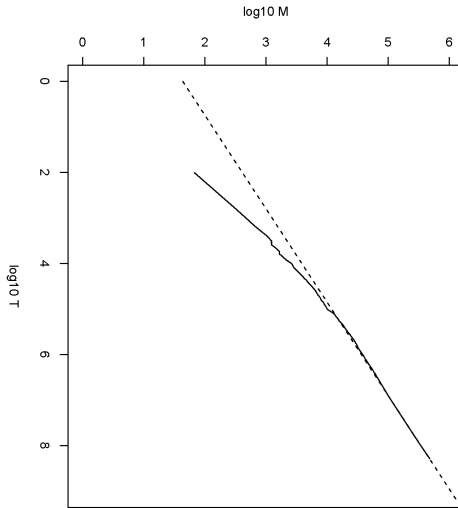
Analysing the Term Vocabulary

- ▶ Given a collection size (the number of tokens) T , can we estimate the vocabulary size M ?
- ▶ Yes, we can using Heaps' law:

$$M = kT^b$$

- ▶ k and b are two parameters, typically in the range of: $30 \leq k \leq 100$ and $b \approx 0.5$
- ▶ For Reuters-RCV1, $k = 44$ and $b = 0.49$, which predicts 38,323 (actual number is 38,365)

Heaps' Law for Reuters-RCV1



Heaps' Law

- ▶ Implications of Heaps' Law:
 - ▶ dictionary size keeps growing with more documents (no maximum will be reached)
 - ▶ dictionary sizes will be quite large for large collections
- ▶ Has been shown empirically for large collections
- ▶ Dictionary compression is important for efficiency

Dictionary Compression

- ▶ The dictionary is small compared to the postings file, but
 - ▶ we want to keep (most of) it in memory
 - ▶ there is competition with other applications for memory
 - ▶ small memory sizes for cell phones or onboard computers
 - ▶ we want fast startup time
- ▶ So compressing the dictionary is useful

Dictionary as Fixed-Width Array

	term	document frequency	pointer to postings list
	a	656,265	→
	aachen	65	→

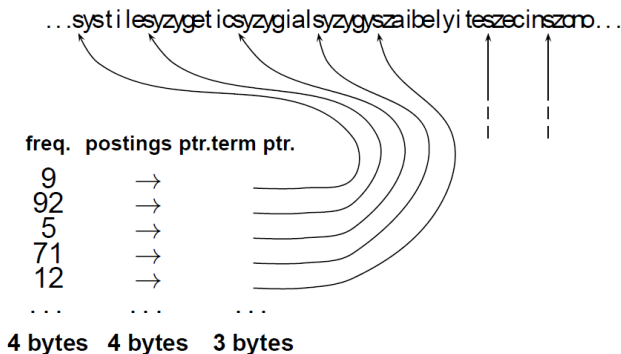
	zulu	221	→
space needed:	20 bytes	4 bytes	4 bytes

- Space for Reuters:
 $(20+4+4)*400,000 = 11.2 \text{ MB}$

Dictionary as Fixed-Width Array

- ▶ This is a bad idea
 - ▶ Most of the bytes in the term column are wasted
 - ▶ 20 bytes are allotted for a term of length 1
 - ▶ We can't handle very long words, e.g.,
"hydrochlorofluorocarbons"
- ▶ The average length of a term in English:
8 characters
 - ▶ How can we use on average 8 characters per term?

Dictionary as a String

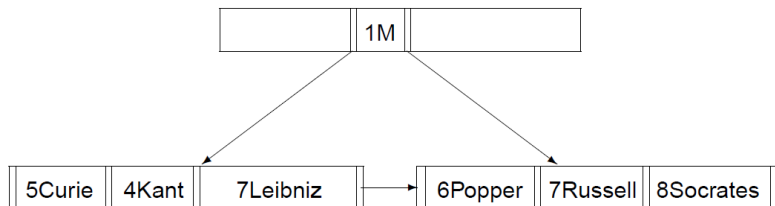


- Concatenate all terms into one big string (and use pointers in dictionary)
- Use binary search to find term

Dictionary as a String

- ▶ While this saves space, it is not very scalable
 - ▶ Once we run out of memory, we have to store (parts of) the dictionary on disk
 - ▶ Switching between main-memory and disk representation is awkward in this scheme
- ▶ We are going to look at prefix B-trees and some optimizations for them
 - ▶ NOTE: we deviate a bit from the textbook here

Prefix B-trees



- ▶ Only the leaves of the trees contain actual terms (+ frequency and postings pointer)
- ▶ Terms use variable space on page (every term preceded by length)
- ▶ Inner nodes use reference keys to separate pages, not actual terms

Prefix B-trees

- ▶ Compared to dictionary as a string, we replace the term pointers with inner nodes
- ▶ This will use slightly more memory, but allows for a much faster search
- ▶ B-trees are very good for disk-based indexing (if we run out of main memory)
- ▶ We can cache the most frequently used parts of the tree in main memory

Front Coding

- ▶ Many entries on a page share the same prefix
- ▶ We can exploit this by using front coding:
 - ▶ 1st number indicates how many letters to re-use from the beginning of previous word
 - ▶ 2nd number states how many letter to add to this
 - ▶ This is followed by the actual letters

word	front coding
automata	0,8,automata
automate	7,1,e
automatic	7,2,ic
automation	8,2,on
automotive	5,5,otive
bat	0,3,bat

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

Key Idea: Store Gaps

- ▶ Each postings list is ordered in the increasing order of docID
 - ▶ information, 8: $\langle 3, 8, 12, 19, 22, 23, 26, 33 \rangle$;
- ▶ It suffices to store *gaps*
 - ▶ information, 8: $\langle 3, 5, 4, 7, 3, 1, 3, 7 \rangle$;
- ▶ The gaps for frequent terms are small.
 - ▶ That means, we can encode small gaps with fewer than 20 bits.

Gap Encoding

	encoding	postings list					
the	docIDs	...	283042		283043	283044	...
	gaps			1		1	...
computer	docIDs	...	283047		283154	283159	...
	gaps			107		5	...
arachnocentric	docIDs	252000	500100				
	gaps	252000	248100				

- ▶ For rare terms, such as arachnocentric, gaps can be quite large
 - ▶ We still need 20 bits to encode them
- ▶ Solution: use *variable length encoding*
 - ▶ Few bits for small gaps
 - ▶ Many bits for large gaps

Variable Byte (VB) Code

- ▶ Used by many commercial/research systems
- ▶ Good low-tech blend of variable-length coding and sensitivity to alignment matches (bit-level codes, see later).
 - ▶ 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: set $c = 0$, encode high-order 7 bits and then use one or more additional bytes to encode the lower order bits using the same algorithm.

VB Code Example

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

- ▶ 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

Bit-Level Encoding

- ▶ You can get even more compression with *bit-level* code
 - ▶ These use variable length bit codes
 - ▶ Have to be *prefix-free*, i.e. no valid codeword is allowed to be the prefix of another (like phone numbers)
- ▶ We are going to look at *Unary Code* and *Gamma Code* (or γ -Code)

Unary Code

- [illegible]

Gamma Code (γ -Code)

- ▶ How many bits do we need to store a gap?
 - ▶ $1=1$ (1 bit), $2=10$ (2 bits), $3=11$ (2 bits),
 $5=101$ (3 bits), $13=1101$ (4 bits), ...
 - ▶ The number of bits to store n : $1 + \log_2 \lfloor n \rfloor$
 - ▶ We don't have any gaps of 0, therefore 1 is the smallest number we have to encode
 - ▶ This means that we always have a leading 1
 - ▶ We can chop off the leading 1, and measure the *length* of the remaining bit-string: $\log_2 \lfloor n \rfloor$
 - ▶ We call the remaining bitstring the *offset*
 - ▶ Example: 13 is 1101,
chop off leading 1 \rightarrow 101:
length = 3, offset = 101

Gamma Code

- ▶ Gamma Code encodes
 - ▶ the length in unary code
 - ▶ the offset in the usual binary code
- ▶ So, for our example:
 - ▶ 13 has a gamma code of 1110101:
1110 for the length, 101 for the offset

More 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		1111111110	11111111	111111110,11111111
1025		11111111110	0000000001	11111111110,0000000001

Comparison

- ▶ So, which code is better?
 - ▶ Here is a comparison for some typical document collections:

method	bits per gap			
	Bible	GNUBib	Comact	TREC
Unary	262	909	487	1918
Gamma	6.51	5.68	4.48	6.63

- ▶ More details (+ another code, Delta Code) in:
 - ▶ I.H. Witten, A. Moffat und T.C. Bell,
"Managing Gigabytes", Morgan Kaufmann, 1999

Comparison

- ▶ Machines have word boundaries: 8, 16, 32 bits
- ▶ Compressing and manipulating at individual bit-granularity
 - ▶ yields better compression
 - ▶ can slow down query processing
- ▶ Variable byte alignment is potentially more efficient to process
 - ▶ Regardless of efficiency, variable byte is conceptually simpler at little additional space cost

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 have ignored positional and frequency information
 - ▶ For this reason, space savings are less in reality
 - ▶ But similar techniques can be used to compress positional information