

Introduction to **Information Retrieval**

CS276: Information Retrieval and Web Search

Christopher Manning and Pandu Nayak

Lecture 4: Index Compression

Last lecture – index construction

- Sort-based indexing
 - Naïve in-memory inversion
 - Blocked Sort-Based Indexing (BSBI)
 - Merge sort is effective for hard disk–based sorting (avoid seeks!)
- Single-Pass In-Memory Indexing (SPIMI)
 - No global dictionary
 - Generate separate dictionary for each block
 - Don't sort postings
 - Accumulate postings in postings lists as they occur
- Distributed indexing using MapReduce
- Dynamic indexing: Multiple indices, logarithmic merge

Today

BRUTUS →

1	2	4	11	31	45	173	174
---	---	---	----	----	----	-----	-----

CAESAR →

1	2	4	5	6	16	57	132	...
---	---	---	---	---	----	----	-----	-----

CALPURNIA →

2	31	54	101
---	----	----	-----

- Collection statistics in more detail (with RCV1)
 - How big will the dictionary and postings be?
- Dictionary compression
- Postings compression

Why compression (in general)?

- Use less disk space
 - Save a little money; give users more space
- Keep more stuff in memory
 - Increases speed
- Increase speed of data transfer from disk to memory
 - [read compressed data | decompress] is faster than [read uncompressed data]
- Premise: Decompression algorithms are fast
 - True of the decompression algorithms we use

Why compression for inverted indexes?

- Dictionary
 - Make it small enough to keep in main memory
 - Make it so small that you can keep some postings lists in main memory too
- Postings file(s)
 - Reduce disk space needed
 - Decrease time needed to read postings lists from disk
 - Large search engines keep a significant part of the postings in memory.
 - Compression lets you keep more in memory
- We will devise various IR-specific compression schemes

Recall Reuters RCV1

■ symbol	statistic	value
■ N	documents	800,000
■ L	avg. # tokens per doc	200
■ M	terms (= 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	7.5
■	non-positional postings	100,000,000

Index parameters vs. what we index

(details *IIR* Table 5.1, p.80)

size of	word types (terms)			non-positional postings			positional postings		
	dictionary			non-positional index			positional index		
	Size (K)	$\Delta\%$	cumul %	Size (K)	$\Delta\%$	cumul %	Size (K)	$\Delta\%$	cumul %
Unfiltered	484			109,971			197,879		
No numbers	474	-2	-2	100,680	-8	-8	179,158	-9	-9
Case folding	392	-17	-19	96,969	-3	-12	179,158	0	-9
30 stopwords	391	-0	-19	83,390	-14	-24	121,858	-31	-38
150 stopwords	391	-0	-19	67,002	-30	-39	94,517	-47	-52
stemming	322	-17	-33	63,812	-4	-42	94,517	0	-52

Exercise: give intuitions for all the '0' entries. Why do some zero entries correspond to big deltas in other columns?

Lossless vs. lossy compression

- Lossless compression: All information is preserved.
 - What we mostly do in IR.
- Lossy compression: Discard some information
- Several of the preprocessing steps can be viewed as lossy compression: case folding, stop words, stemming, number elimination.
- Chapter 7: Prune postings entries that are unlikely to turn up in the top k list for any query.
 - Almost no loss of quality in top k list.

Vocabulary size vs. collection size

- How big is the term vocabulary?
 - That is, how many distinct words are there?
- Can we assume an upper bound?
 - Not really: At least $70^{20} = 10^{37}$ different words of length 20
- In practice, the vocabulary will keep growing with the collection size
 - Especially with Unicode 😊

Vocabulary size vs. collection size

- Heaps' law: $M = kT^b$
- M is the size of the vocabulary, T is the number of tokens in the collection
- Typical values: $30 \leq k \leq 100$ and $b \approx 0.5$
- In a log-log plot of vocabulary size M vs. T , Heaps' law predicts a line with slope about $\frac{1}{2}$
 - It is the simplest possible (linear) relationship between the two in log-log space
 - $\log M = \log k + b \log T$
 - An empirical finding (“empirical law”)

Heaps' Law

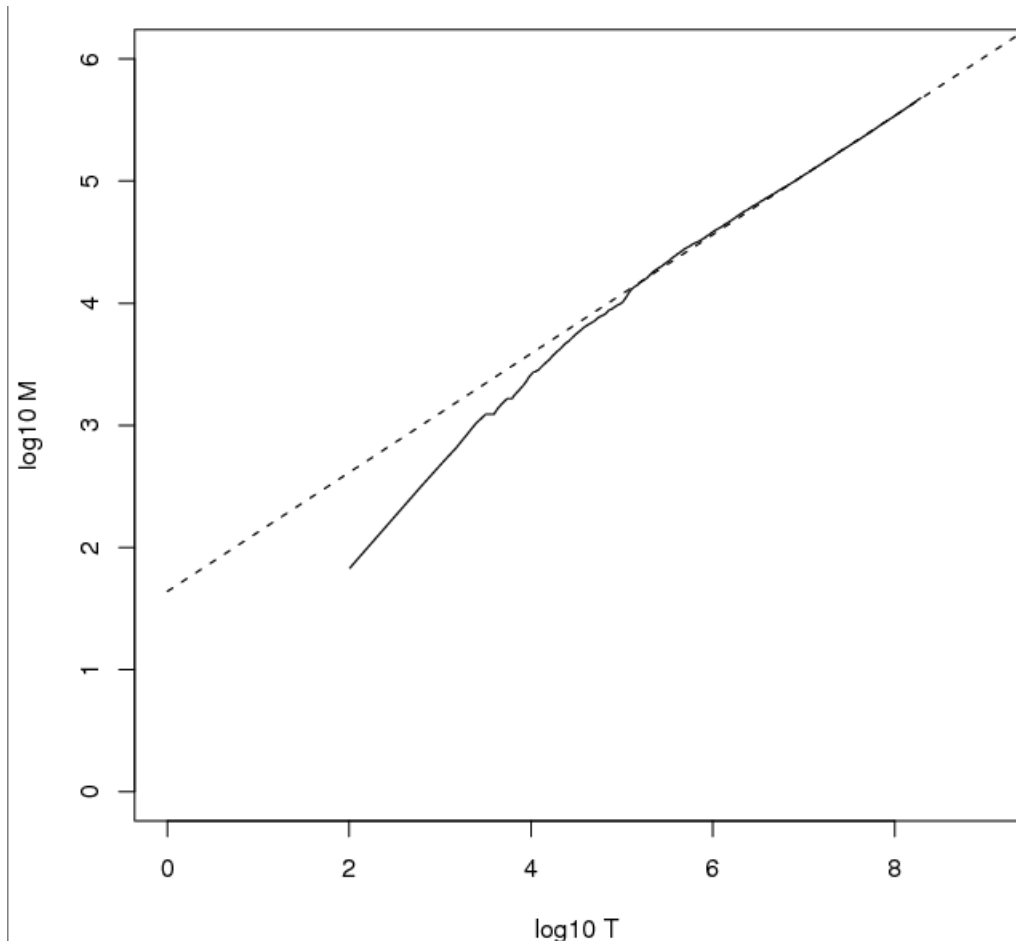
For RCV1, the dashed line
 $\log_{10} M = 0.49 \log_{10} T + 1.64$
is the best least squares fit.

Thus, $M = 10^{1.64} T^{0.49}$ so $k = 10^{1.64} \approx 44$ and $b = 0.49$.

Good empirical fit for
Reuters RCV1 !

For first 1,000,020 tokens,
law predicts 38,323 terms;
actually, 38,365 terms

Fig 5.1 p81



Exercises

- What is the effect of including spelling errors, vs. automatically correcting spelling errors on Heaps' law?
- Compute the vocabulary size M for this scenario:
 - Looking at a collection of web pages, you find that there are 3000 different terms in the first 10,000 tokens and 30,000 different terms in the first 1,000,000 tokens.
 - Assume a search engine indexes a total of 20,000,000,000 (2×10^{10}) pages, containing 200 tokens on average
 - What is the size of the vocabulary of the indexed collection as predicted by Heaps' law?

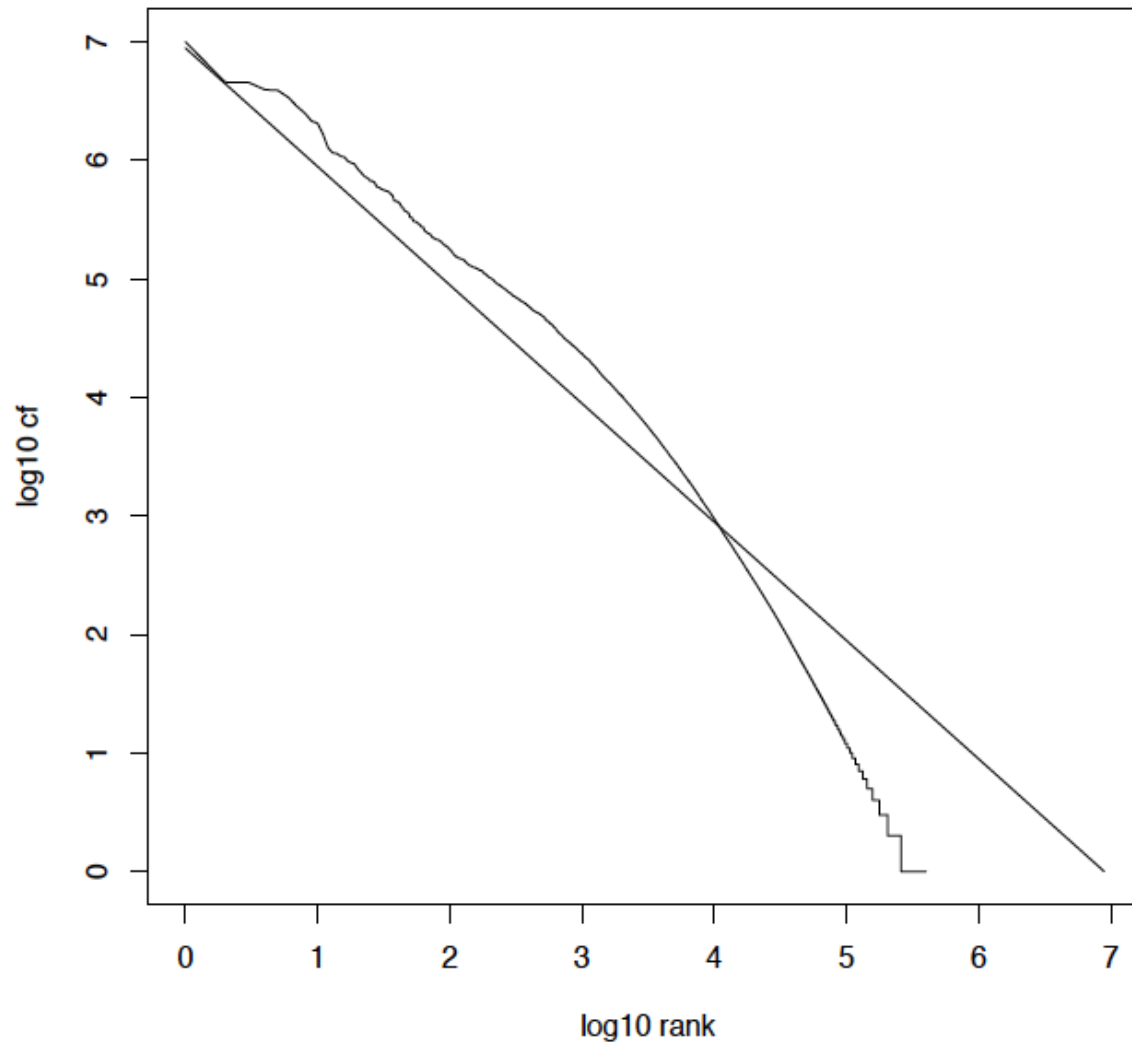
Zipf's law

- Heaps' law gives the vocabulary size in collections.
- We also study the relative frequencies of terms.
- 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 proportional to $1/i$.
- $cf_i \propto 1/i = K/i$ where K is a normalizing constant
- cf_i is collection frequency: the number of occurrences of the term t_i in the collection.

Zipf consequences

- If the most frequent term (*the*) occurs cf_1 times
 - then the second most frequent term (*of*) occurs $cf_1/2$ times
 - the third most frequent term (*and*) occurs $cf_1/3$ times ...
- Equivalent: $cf_i = K/i$ where K is a normalizing factor, so
 - $\log cf_i = \log K - \log i$
 - Linear relationship between $\log cf_i$ and $\log i$
- Another power law relationship

Zipf's law for Reuters RCV1



Compression

- Now, we will consider compressing the space for the dictionary and postings. We'll do:
 - Basic Boolean index only
 - No study of positional indexes, etc.
- But these ideas can be extended
- We will consider compression schemes

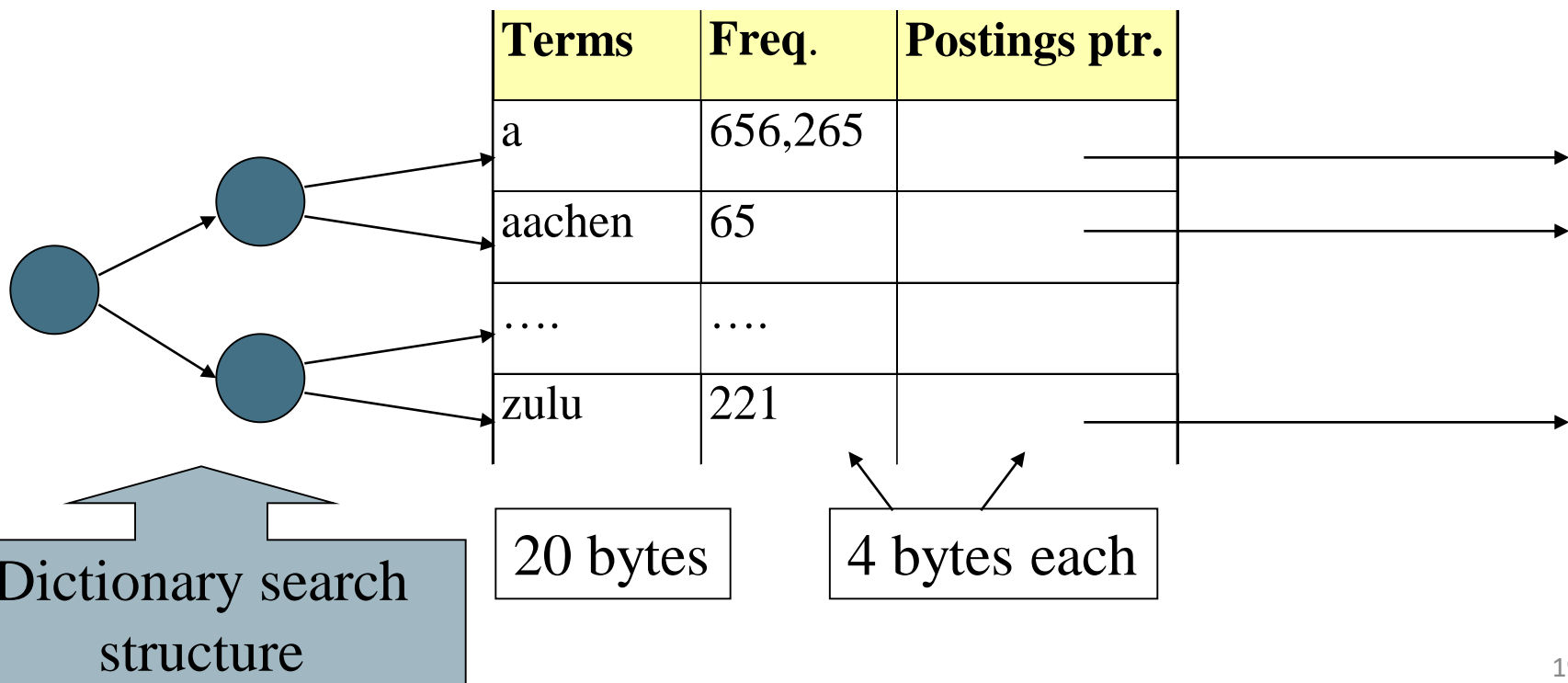
DICTIONARY COMPRESSION

Why compress the dictionary?

- Search begins with the dictionary
- We want to keep it in memory
- Memory footprint competition with other applications
- Embedded/mobile devices may have very little memory
- Even if the dictionary isn't in memory, we want it to be small for a fast search startup time
- So, compressing the dictionary is important

Dictionary storage – naïve version

- Array of fixed-width entries
 - ~400,000 terms; 28 bytes/term = 11.2 MB.

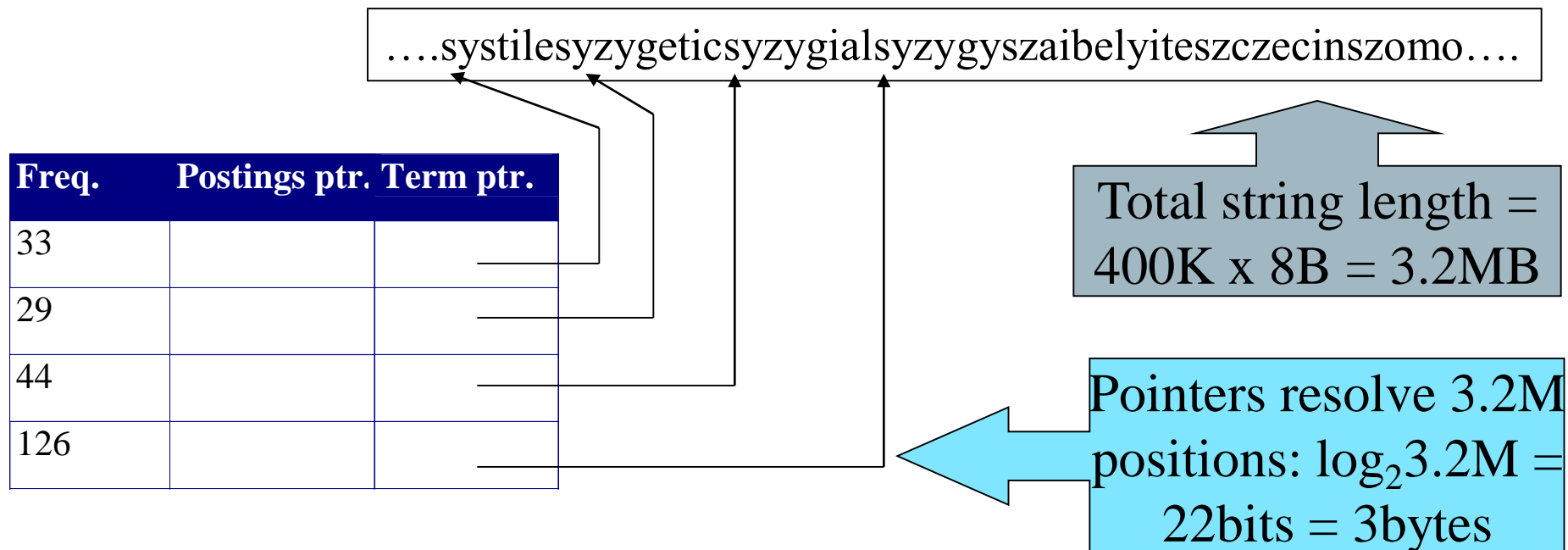


Fixed-width terms are wasteful

- Most of the bytes in the **Term** column are wasted – we allot 20 bytes for 1 letter terms.
 - And we still can't handle *supercalifragilisticexpialidocious* or *hydrochlorofluorocarbons*.
- Written English averages ~4.5 characters/word.
 - Exercise: Why is/isn't this the number to use for estimating the dictionary size?
- Ave. dictionary word in English: ~8 characters
 - How do we use ~8 characters per dictionary term?
- Short words dominate token counts but not type average.

Compressing the term list: Dictionary-as-a-String

- Store dictionary as a (long) string of characters:
 - Pointer to next word shows end of current word
 - Hope to save up to 60% of dictionary space



Space for dictionary as a string

- 4 bytes per term for Freq.
 - 4 bytes per term for pointer to Postings.
 - 3 bytes per term pointer
 - Avg. 8 bytes per term in term string
 - 400K terms \times 19 \Rightarrow 7.6 MB (against 11.2MB for fixed width)
- } Now avg. 11 bytes/term, not 20.

Blocking

- Store pointers to every k th term string.
 - Example below: $k=4$.
- Need to store term lengths (1 extra byte)

....**7***systile***9***syzygetic***8***syzygial***6***syzygy***11***szaibelyite***8***szczecin***9***szomo*....

Freq.	Postings ptr.	Term ptr.
33		
29		
44		
126		
7		

} Save 9 bytes
} on 3
} pointers.

← Lose 4 bytes on
term lengths.

Blocking Net Gains

- Example for block size $k = 4$
- Where we used 3 bytes/pointer without blocking
 - $3 \times 4 = 12$ bytes,

now we use $3 + 4 = 7$ bytes.

Shaved another ~ 0.5 MB. This reduces the size of the dictionary from 7.6 MB to 7.1 MB.

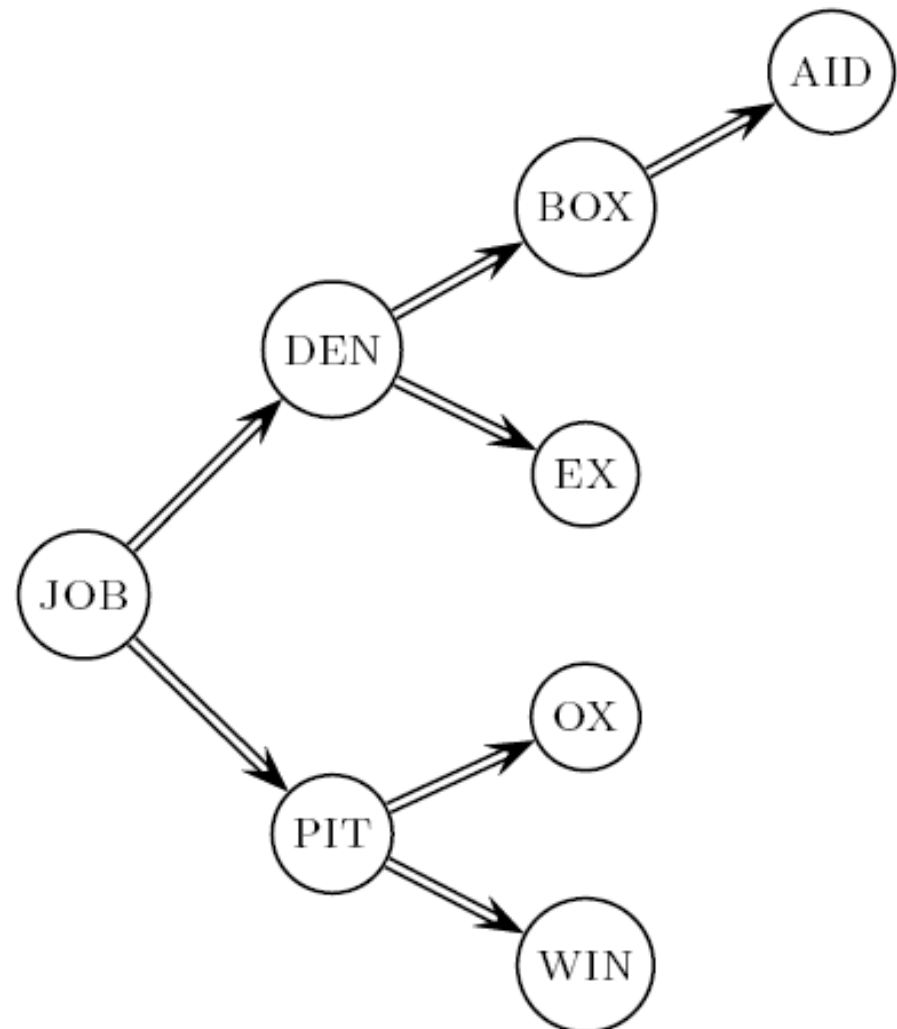
We can save more with larger k .

Question: Why not go with larger k ?

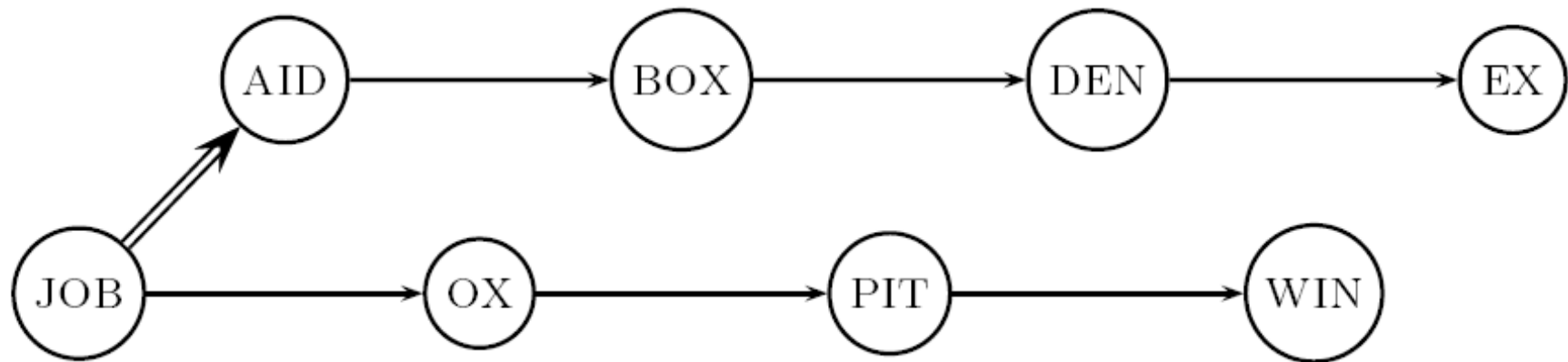
Dictionary search without blocking

- Assuming each dictionary term equally likely in query (not really so in practice!), average number of comparisons = $(1+2\cdot 2+4\cdot 3+4)/8 \sim 2.6$

Exercise: what if the frequencies of query terms were non-uniform but known, how would you structure the dictionary search tree?



Dictionary search with blocking



- Binary search down to 4-term block;
 - Then linear search through terms in block.
- Blocks of 4 (binary tree), avg. =
 $(1+2\cdot 2+2\cdot 3+2\cdot 4+5)/8 = 3$ compares

Exercises

- Estimate the space usage (and savings compared to 7.6 MB) with blocking, for block sizes of $k = 4, 8$ and 16 .
- Estimate the impact on search performance (and slowdown compared to $k=1$) with blocking, for block sizes of $k = 4, 8$ and 16 .

Front coding

- Front-coding:
 - Sorted words commonly have long common prefix – store differences only
 - (for last $k-1$ in a block of k)

8*automata***8***automate***9***automatic***10***automation*

→ **8***automat****a**1♦**e**2♦**i**3♦**ion**

Encodes prefix *automat*

Extra length
beyond *automat*.

Begins to resemble general string compression. 28

RCV1 dictionary compression summary

Technique	Size in MB
Fixed width	11.2
Dictionary-as-String with pointers to every term	7.6
+ blocking, $k = 4$	7.1
+ blocking + front coding	5.9

POSTINGS COMPRESSION

Postings compression

- The postings file is much larger than the dictionary, factor of at least 10, often over 100 times larger
- 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 far fewer than 20 bits per docID.

Postings: two conflicting forces

- A term like ***arachnocentric*** occurs in maybe one doc out of a million – we would like to store this posting using $\log_2 1M \approx 20$ bits.
- A term like ***the*** occurs in virtually every doc, so 20 bits/posting $\approx 2MB$ is too expensive.
 - Prefer 0/1 bitmap vector in this case ($\approx 100K$)

Gap encoding of postings file entries

- We store the list of docs containing a term in increasing order of docID.
 - **computer**: 33,47,154,159,202 ...
- Consequence: it suffices to store *gaps*.
 - 33,14,107,5,43 ...
- Hope: most gaps can be encoded/stored with far fewer than 20 bits.
 - Especially for common words

Three postings entries

	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 *arachnocentric*, we will use ~ 20 bits/gap entry.
 - For *the*, we will use ~ 1 bit/gap entry.
- If the average gap for a term is G , we want to use $\sim \log_2 G$ bits/gap entry.
- Key challenge: encode every integer (gap) with about as few bits as needed for that integer.
- This requires a *variable length encoding*
- Variable length codes achieve this by using short codes for small numbers

- Unary code for 40 is

- Unary code for 80 is:

- This doesn't look promising, but....

- 36

Gamma codes

- We can compress better with bit-level codes
 - The Gamma code is the best known of these.
- Represent a gap G as a pair *length* and *offset*
- *offset* is G in binary, with the leading bit cut off
 - For example $13 \rightarrow 1101 \rightarrow 101$
- *length* is the length of offset
 - For 13 (offset 101), this is 3.
- We encode *length* with *unary code*: 1110.
- Gamma code of 13 is the concatenation of *length* and *offset*: 1110101

Gamma code examples

number	length	offset	γ -code
0			none
1	0		0
2	10	0	10,0
3	10	1	10,1
4	110	00	110,00
9	1110	001	1110,001
13	1110	101	1110,101
24	11110	1000	11110,1000
511	111111110	11111111	111111110,11111111
1025	11111111110	0000000001	11111111110,0000000001

Reminder: bitwise operations

- For compression, you need to use bitwise operators

CS107 Schedule Assignments Labs Gradebook Resources Getting Help

Computer Organization & Systems

Week 2

Lecture 3 (Mon 4/8): Bits and Bitwise Operators

We'll dive further into bits and bytes, and how to manipulate them using bitwise operators.

[Lecture 3 Slides](#)
B&O Ch 2.1

In: assign0
Out: assign1

- Python (and most everything else):
 - & bitwise and; | bitwise or; ^ bitwise xor; ~ ones complement
 - << left shift bits, >> right shift; LACKS >>> zero fill right shift
 - Recipes:
 - Extract 7 bits: `a & 0x7f00 >> 8` ; if take high-order bit add: `& 0x7f`
 - Combine 3 5-bit numbers: `a | (b << 5) | (c << 10)`
 - Lookup tables rather than decoding can be faster, yet still small

Gamma code properties

- G is encoded using $2 \lfloor \log G \rfloor + 1$ bits
 - Length of offset is $\lfloor \log G \rfloor$ bits
 - Length of length is $\lfloor \log G \rfloor + 1$ bits
- All gamma codes have an odd number of bits
- Almost within a factor of 2 of best possible, $\log_2 G$
- Gamma code is uniquely prefix-decodable, like VB
- Gamma code can be used for any distribution
 - Optimal for $P(n) \approx 1/(2n^2)$
- Gamma code is parameter-free

Gamma seldom used in practice

- Machines have word boundaries – 8, 16, 32, 64 bits
 - Operations that cross word boundaries are slower
- Compressing and manipulating at the granularity of bits can be too slow
- All modern practice is to use byte or word aligned codes
 - Variable byte encoding is a faster, conceptually simpler compression scheme, with decent compression

Variable Byte (VB) codes

- For a gap value G , we want to use close to the fewest bytes needed to hold $\log_2 G$ bits
- Begin with one byte to store G and dedicate 1 bit in it to be a continuation bit c
- If $G \leq 127$, binary-encode it in the 7 available bits and set $c = 1$
- Else encode G 's lower-order 7 bits and then use 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 for the other bytes $c = 0$.

Example

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

Postings stored as the byte concatenation

000001101011100010000101000011010000110010110001

Key property: VB-encoded postings are uniquely prefix-decodable.

For a small gap (5), VB uses a whole byte.


RCV1 compression

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)	3,600.0
collection (text)	960.0
Term-doc 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

Other variable unit codes

- Variable byte codes are used by many real systems
 - Good low-tech blend of variable-length coding and sensitivity to computer memory alignment matches
- Byte alignment wastes space if you have many small gaps – as gap encoding often makes
- More modern work mainly uses the ideas:
 - Be word aligned (32 or 64 bits; even faster)
 - Encode several gaps at the same time
 - Often assume a maximum gap size, perhaps with an escape

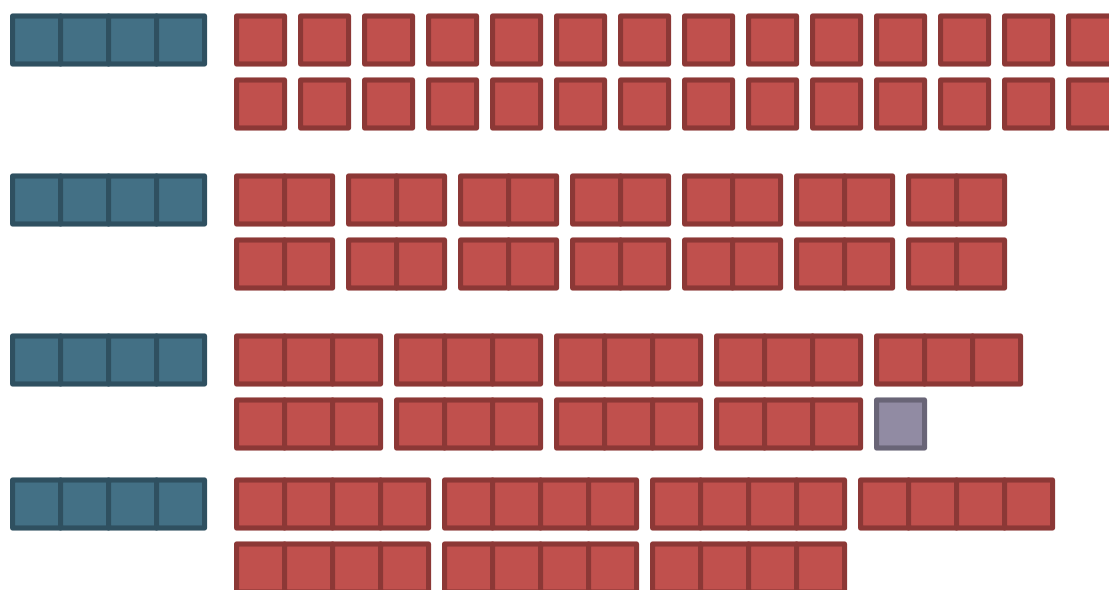
Group Variable Integer code

- Used by Google around turn of millennium....
 - Jeff Dean, keynote at WSDM 2009 and presentations at CS276
- Encodes 4 integers in blocks of size 5–17 bytes
- First byte: four 2-bit binary length fields
-  , $L_j \in \{1, 2, 3, 4\}$
- Then, $L_1 + L_2 + L_3 + L_4$ bytes (between 4–16) hold 4 numbers
 - Each number can use 8/16/24/32 bits. Max gap length ~4 billion
- It was suggested that this was about twice as fast as VB encoding
 - Decoding gaps is much simpler – no bit masking
 - First byte can be decoded with lookup table or switch

Simple-9 [Anh & Moffat, 2004]

A word-aligned, multiple number encoding scheme

How can we store several numbers in 32 bits with a format selector?



Simple9 Encoding Scheme [Anh & Moffat, 2004]

- Encoding block: 4 bytes (32 bits)
- Most significant nibble (4 bits) describe the layout of the 28 other bits as follows:

Layout (4 bits)	n numbers of b bits each $n * b \leq 28$
--------------------	---

 - 0: a single 28-bit number
 - 1: two 14-bit numbers
 - 2: three 9-bit numbers (and one spare bit)
 - 3: four 7-bit numbers
 - 4: five 5-bit numbers (and three spare bits)
 - 5: seven 4-bit numbers
 - 6: nine 3-bit numbers (and one spare bit)
 - 7: fourteen two-bit numbers
 - 8: twenty-eight one-bit numbers
- Simple16 is a variant with 5 additional (uneven) configurations
- Efficiently decoded with hand-coded decoder, using bit masks
- Extended Simple Family – idea applies to 64-bit words, etc.

Index compression summary

- We can now create an index for highly efficient Boolean retrieval that is very space efficient
- Only 4% of the total size of the collection
- Only 10-15% of the total size of the text in the collection

- We've ignored positional information
- Hence, space savings are less for indexes used in practice
 - But techniques substantially the same

Resources for today's lecture

- *IIR 5*
- *MG 3.3, 3.4.*
- F. Scholer, H.E. Williams and J. Zobel. 2002. Compression of Inverted Indexes For Fast Query Evaluation. *Proc. ACM-SIGIR 2002*.
 - Variable byte codes
- V. N. Anh and A. Moffat. 2005. Inverted Index Compression Using Word-Aligned Binary Codes. *Information Retrieval 8*: 151–166.
 - Word aligned codes