

# Introduction to **Information Retrieval**

## Lecture 5: Index Compression

# Last lecture – index construction

---

- Sort-based indexing
  - Naïve in-memory inversion
  - Blocked Sort-Based Indexing
    - Merge sort is effective for disk-based sorting (avoid seeks!)
- Single-Pass In-Memory Indexing
  - 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
  - Saves a little money
- 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.
- Chap/Lecture 7: Prune postings entries that are unlikely to turn up in the top  $k$  list for any query.
  - Almost no loss quality for top  $k$  list.



# Vocabulary 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 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 relationship between the two in log-log space
  - An empirical finding ("empirical law")

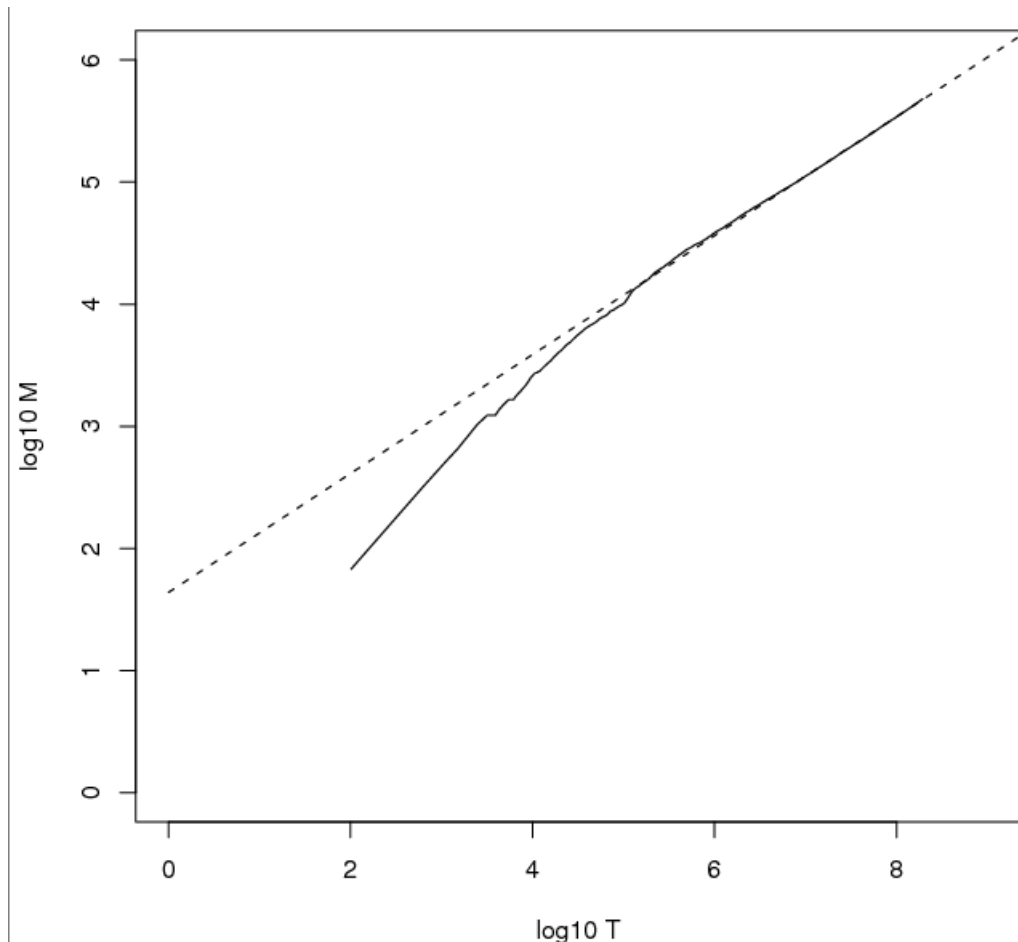
# Heaps' Law

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 ( $2 \times 10^{10}$ ) 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

---

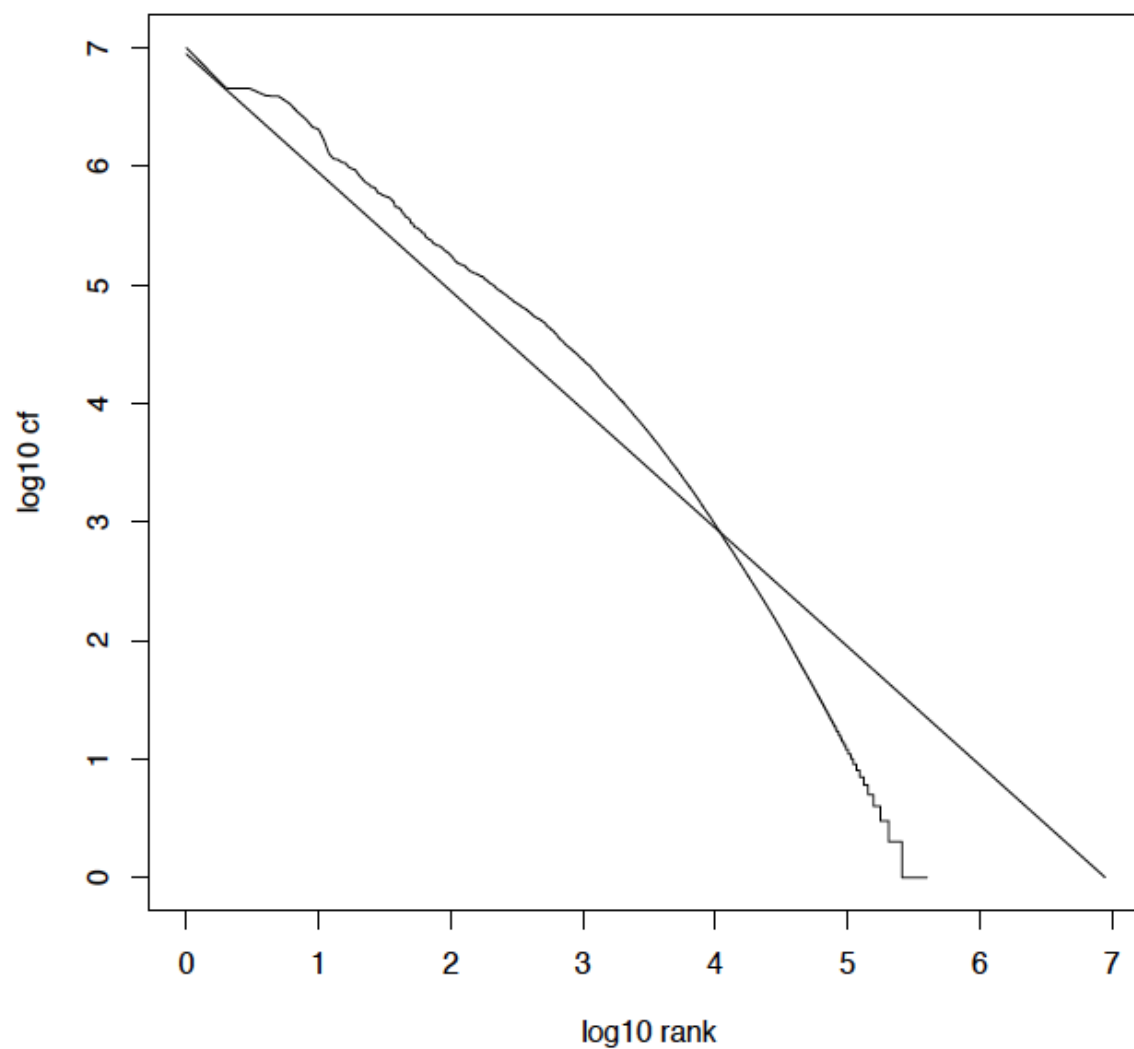
- Heaps' law gives the vocabulary size in collections.
- We also study the relative frequencies of terms.
- In natural language, there are a few very frequent terms and very many very rare terms.
- Zipf's law: The  $i$ th most frequent term has frequency proportional to  $1/i$ .
- $cf_i \propto 1/i = K/i$  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
  - Basic Boolean index only
  - No study of positional indexes, etc.
  - We will consider compression schemes



# DICTIONARY COMPRESSION

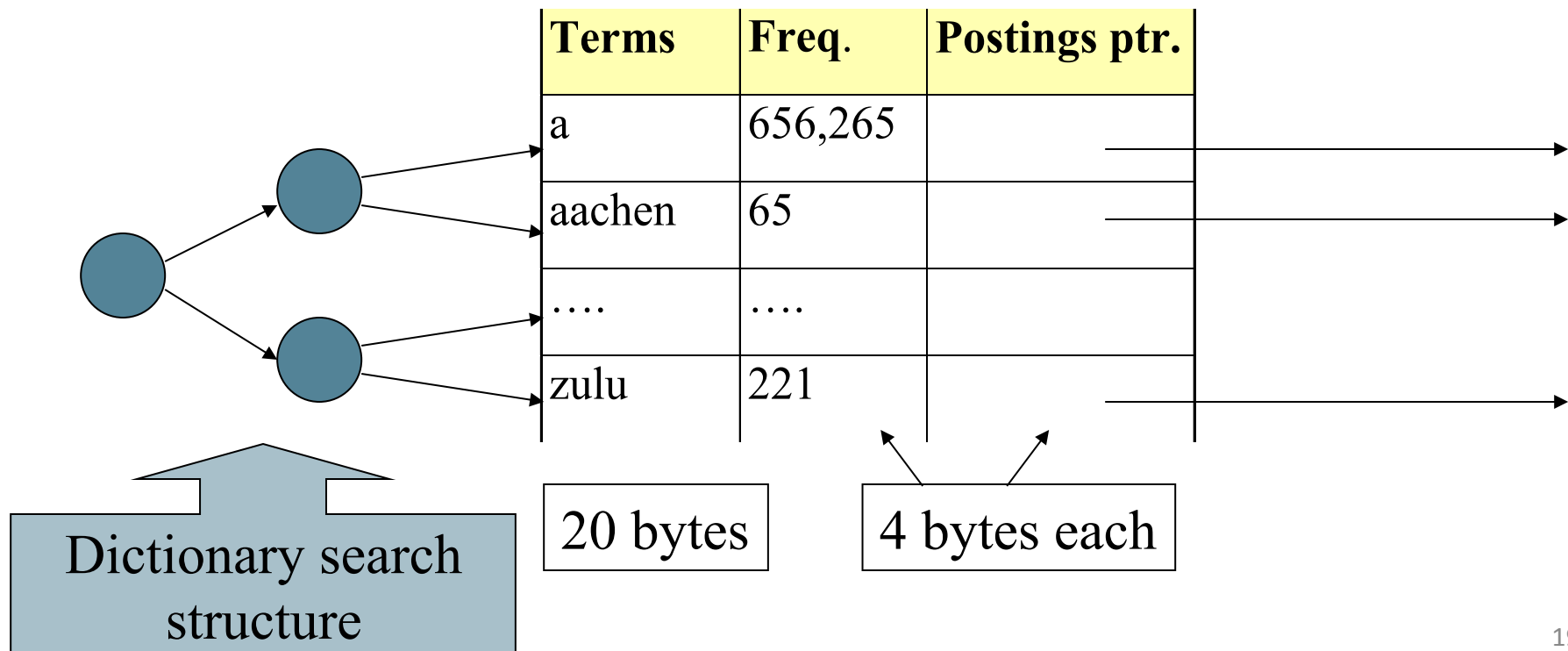
# Why compress the dictionary?

---

- 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 - first cut

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



# Fixed-width terms are wasteful

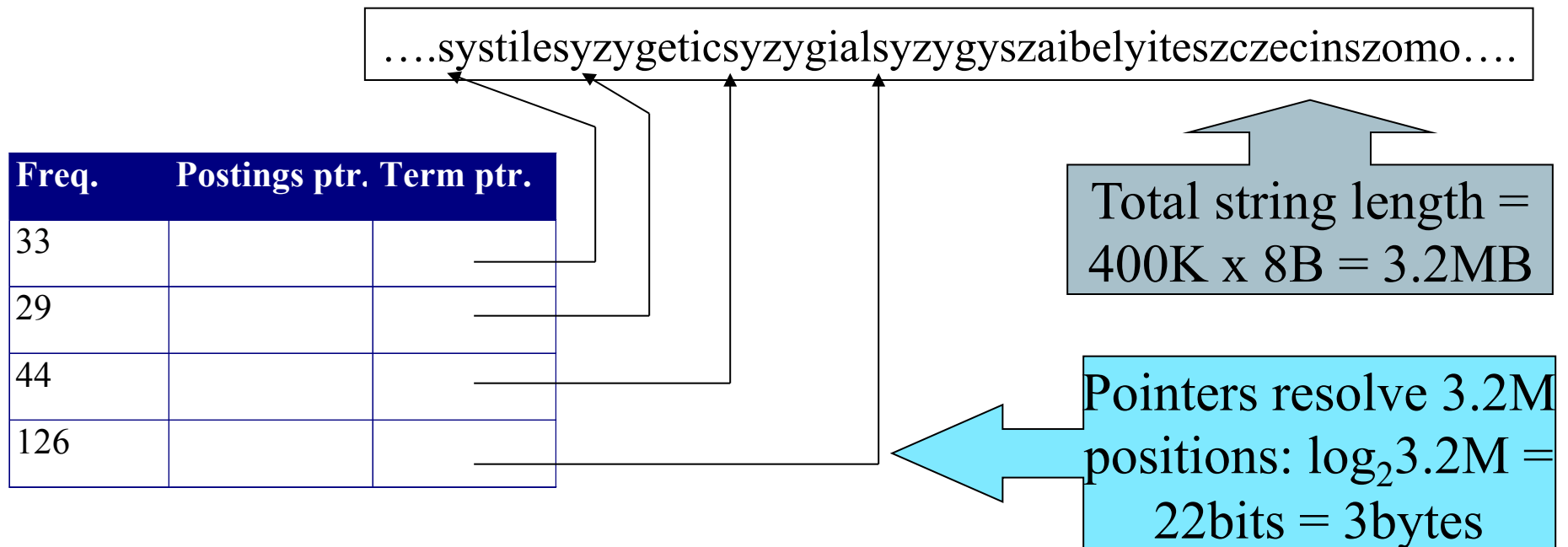
---

- 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 x 19 → 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.

# Net

---

- 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  $\sim 0.5$ MB. This reduces the size of the dictionary from 7.6 MB to 7.1 MB.

We can save more with larger  $k$ .

Why not go with larger  $k$ ?



# Exercise

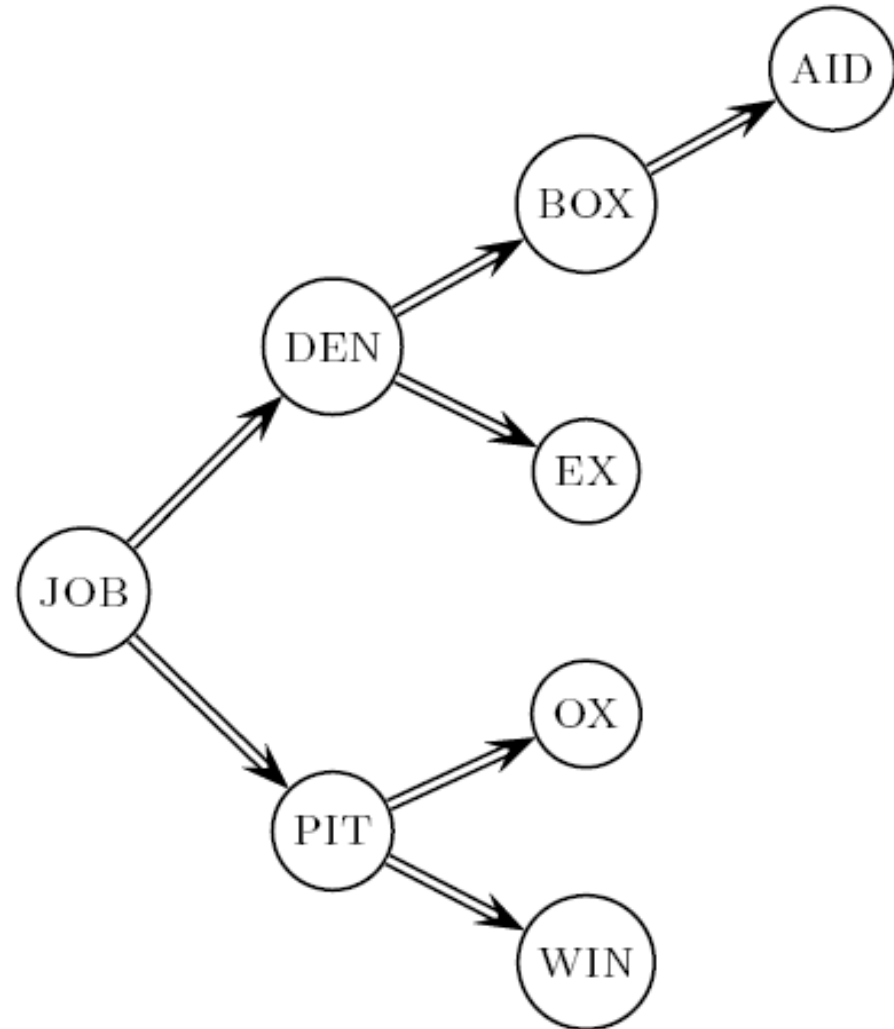
---

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

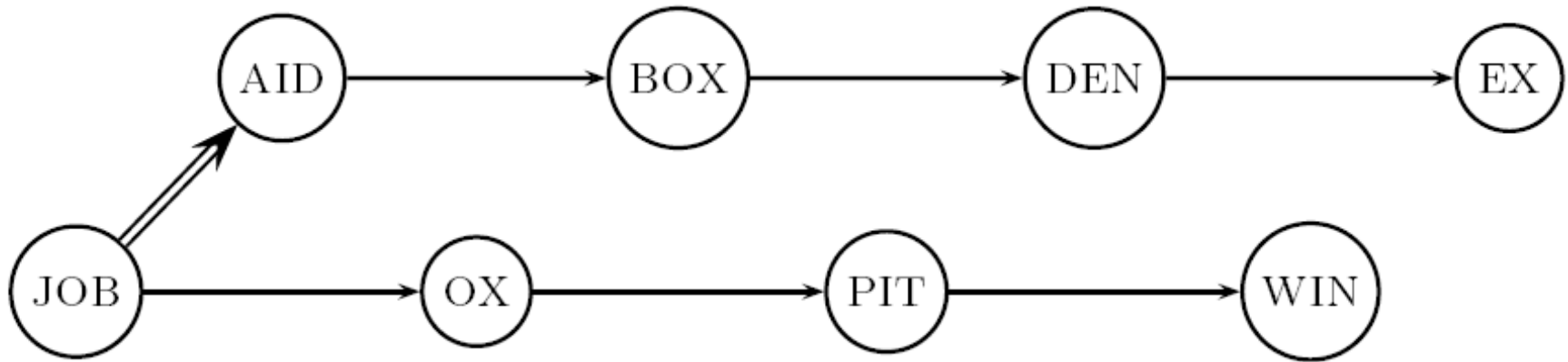
## Dictionary search without blocking

- 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

# Exercise

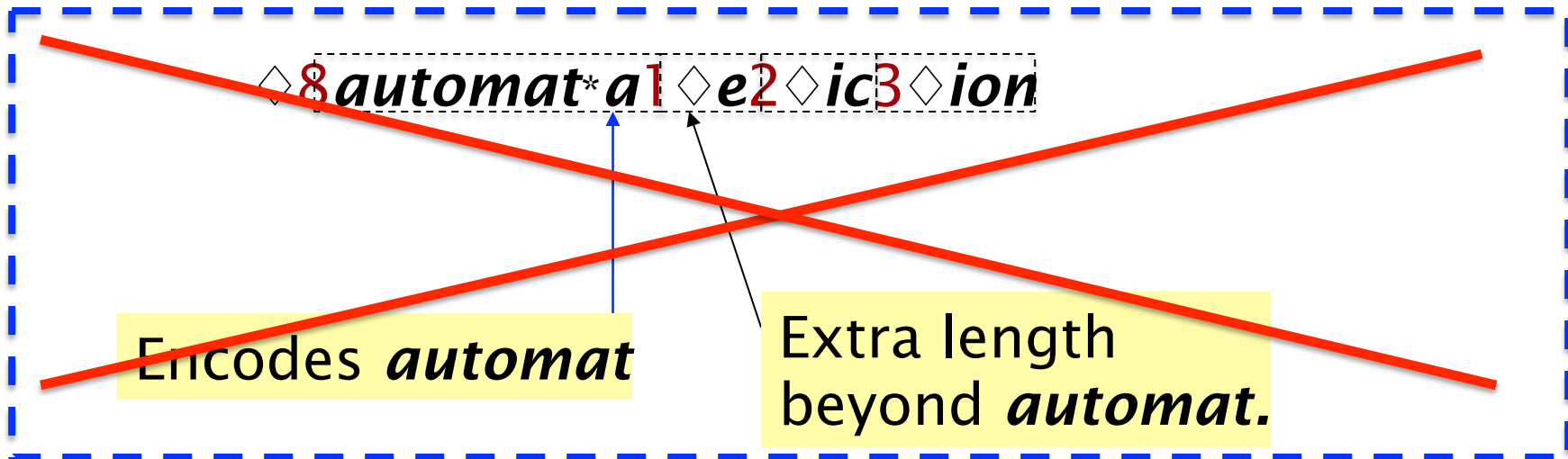
---

- 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*



Begins to resemble general string compression. 29

# Front Encoding [Witten, Moffat, Bell]

- Complete front encoding
  - (prefix-len, suffix-len, suffix)
- Partial 3-in-4 front encoding
  - No encoding/compression for the first string in a block
  - Enables binary search

Assume  
previous  
string is  
“auto”



String	Complete Front Encoding	Partial 3-in-4 Front Encoding
8, automata	4, 4, mata	, 8, automata
8, automate	7, 1, e	7, 1, e
9, automatic	7, 2, ic	7, 2, ic
10, automation	8, 2, on	8, , on

# RCV1 dictionary compression summary

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

# POSTINGS COMPRESSION



# 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.

## 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  $\log_2 1M \sim 20$  bits.
- A term like ***the*** occurs in virtually every doc, so 20 bits/posting is too expensive.
  - Prefer 0/1 bitmap vector in this case

# Postings file entry

---

- 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.

# 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  $\sim 20$  bits/gap entry.
  - For *the*, we will use  $\sim 1$  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

# 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  $c = 0$ .

Hex(824)=0x0338

Hex(214577)=0x00034631

# Example

docIDs	824	829	215406
gaps		5	214577
VB code	0000110 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.

## Other variable unit codes

---

- Instead of bytes, we can also use a different “unit of alignment”: 32 bits (words), 16 bits, 4 bits (nibbles).
- Variable byte alignment wastes space if you have many small gaps – nibbles do better in such cases.
- Variable byte codes:
  - Used by many commercial/research systems
  - Good low-tech blend of variable-length coding and sensitivity to computer memory alignment matches (vs. bit-level codes, which we look at next).
- There is also recent work on word-aligned codes that pack a variable number of gaps into one word (e.g., simple9)



# Simple9

- Encodes as many gaps as possible in one DWORD
- 4 bit selector + 28 bit data bits
  - Encodes 9 possible ways to “use” the data bits

Selector	# of gaps encoded	Len of each gap encoded	Wasted bits
0000	28	1	0
0001	14	2	0
0010	9	3	1
0011	7	4	0
0100	5	5	3
0101	4	7	0
0110	3	9	1
0111	2	14	0
1000	1	28	0

# Unary code

- Represent  $n$  as  $n$  1s with a final 0.
- Unary code for 3 is 1110.

- Unary code for 40 is

**110.**

- Unary code for 80 is:

```
1111111111111111111111111111111111111111111111111  
111111111111111111111111111111111111111111110
```

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

# Bit-Aligned Codes

---

- Breaks between encoded numbers can occur after any bit position
- *Unary* code
  - Encode  $k$  by  $k$  1s followed by 0
  - 0 at end makes code unambiguous

Number	Code
0	0
1	10
2	110
3	1110
4	11110
5	111110

# Unary and Binary Codes

---

- Unary is very efficient for small numbers such as 0 and 1, but quickly becomes very expensive
  - 1023 can be represented in 10 binary bits, but requires 1024 bits in unary
- Binary is more efficient for large numbers, but it may be ambiguous

# Elias-γ Code

- To encode a number  $k$ , compute

- $k_d = \lfloor \log_2 k \rfloor$

unary

- $k_r = k - 2^{\lfloor \log_2 k \rfloor}$

binary

- $k_d$  is number of **binary** digits, encoded in **unary**

Number ( $k$ )	$k_d$	$k_r$	Code
1	0	0	0
2	1	0	10 0
3	1	1	10 1
6	2	2	110 10
15	3	7	1110 111
16	4	0	11110 0000
255	7	127	11111110 1111111
1023	9	511	1111111110 111111111

# Elias- $\delta$ Code

---

- Elias- $\gamma$  code uses no more bits than unary, many fewer for  $k > 2$ 
  - 1023 takes 19 bits instead of 1024 bits using unary
- In general, takes  $2\lfloor \log_2 k \rfloor + 1$  bits
- To improve coding of large numbers, use Elias- $\delta$  code
  - Instead of encoding  $k_d$  in unary, we encode  $k_d + 1$  using Elias- $\gamma$
  - Takes approximately  $2 \log_2 \log_2 k + \log_2 k$  bits

# Elias- $\delta$ Code

- Split  $(k_d + 1)$  into:

$$k_{dd} = \lfloor \log_2(k_d + 1) \rfloor$$

$$k_{dr} = (k_d + 1) - 2^{\lfloor \log_2(k_d + 1) \rfloor}$$

- encode  $k_{dd}$  in unary,  $k_{dr}$  in binary, and  $k_r$  in binary

Number ( $k$ )	$k_d$	$k_r$	$k_{dd}$	$k_{dr}$	Code
1	0	0	0	0	0
2	1	0	1	0	10 0 0
3	1	1	1	0	10 0 1
6	2	2	1	1	10 1 10
15	3	7	2	0	110 00 111
16	4	0	2	1	110 01 0000
255	7	127	3	0	1110 000 1111111
1023	9	511	3	2	1110 010 111111111

```

#
# Generating Elias-gamma and Elias-delta codes in Python
#

import math

def unary_encode(n):
    return "1" * n + "0"

def binary_encode(n, width):
    r = ""
    for i in range(0,width):
        if ((1<<i) & n) > 0:
            r = "1" + r
        else:
            r = "0" + r
    return r

def gamma_encode(n):
    logn = int(math.log(n,2))
    return unary_encode( logn ) + " " + binary_encode(n, logn)

def delta_encode(n):
    logn = int(math.log(n,2))
if n == 1:
    return "0"
else:
    loglog = int(math.log(logn+1,2))
    residual = logn+1 - int(math.pow(2, loglog))
    return unary_encode( loglog ) + " " + binary_encode( residual, loglog ) + " " + binary_encode(n, logn)

if __name__ == "__main__":
    for n in [1,2,3, 6, 15,16,255,1023]:
        logn = int(math.log(n,2))
        loglogn = int(math.log(logn+1,2))
        print n, "d_r", logn
        print n, "d_dd", loglogn
        print n, "d_dr", logn + 1 - int(math.pow(2,loglogn))
        print n, "delta", delta_encode(n)
        #print n, "gamma", gamma_encode(n)
        #print n, "binary", binary_encode(n)

```



# 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
- 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 slow
- Variable byte encoding is aligned and thus potentially more efficient
- Regardless of efficiency, variable byte is conceptually simpler at little additional space cost

# Shannon Limit

---

- Is it possible to derive codes that are optimal (under certain assumptions)?
- What is the optimal average code length for a code that encodes each integer (gap lengths) independently?
- Lower bounds on average code length: Shannon entropy
  - $H(X) = - \sum_{x=1}^n \Pr[X=x] \log \Pr[X=x]$
- Asymptotically optimal codes (finite alphabets): arithmetic coding, Huffman codes

How to design an optimal code for geometric distribution?

# Global Bernoulli Model

- Assumption: term occurrence are Bernoulli events
- Notation:
  - $n$ : # of documents,  $m$ : # of terms in vocabulary
  - $N$ : total # of (unique) occurrences
- Probability of a term  $t_j$  occurring in document  $d_i$ :  $p = N/nm$
- Each term-document occurrence is an independent event
- Probability of a gap of length  $x$  is given by the [geometric distribution](#)  $\Pr[X = x] = (1 - p)^{x-1} \cdot p$

It can also be deemed as a generalization of the unary code.

# Golomb Code

- Golomb Code (Golomb 1966): highly efficient way to design optimal Huffman-style code for geometric distribution
  - Parameter  $b$
  - For given  $x \geq 1$ , computer integer quotient  $q = \lfloor (x-1)/b \rfloor$
  - and remainder  $r = (x-1) - q \cdot b$
- Assume  $b = 2^k$ 
  - Encode  $q$  in unary, followed by  $r$  coded in binary
  - A bit complicated if  $b \neq 2^k$ . See wikipedia.
- First step:  $(q+1)$  bits
- Second step:  $\log(b)$  bits

# Golomb Code & Rice Code

---

- How to determine optimal  $b^*$ ?
- Select minimal  $b$  such that

$$(1-p)^b + (1-p)^{b+1} \leq 1$$

- Result due to Gallager and Van Voorhis 1975:  
generates an optimal prefix code for geometric distribution
- Small  $p$  approximation:  
$$b^* \approx \ln 2 / p = 0.69 \cdot \text{avg\_val}$$
- Rice code: only allow  $b = 2^k$

# Local Bernoulli Model

---

- If length of posting lists is known, then a Bernoulli model on each individual inverted list can be used
- Frequent words are coded with smaller  $b$ , infrequent words with larger  $b$
- Term frequency need to be encoded (use gamma-code)
- Local Bernoulli outperforms global Bernoulli model in practice (method of practice!)

# 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, g-encoded	101.0



# Google's Indexing Choice

---

- Index shards partition by doc, multiple replicates
- Disk-resident index
  - Use outer parts of the disk
  - Use different compression methods for different fields: Rice<sub>k</sub> (a special kind of Golomb code) for gaps, and Gamma for positions.
- In-memory index
  - All positions; No docid
    - Keep track of document boundaries
  - Group-variant encoding
    - Fast to decode

Source: [Jeff Dean's WSDM 2009 Keynote](#)

## Other details

---

- $\text{Gap} = \text{docid}_n - \text{docid}_{n-1} - 1$
- $\text{Freq} = \text{freq} - 1$
- $\text{Pos\_Gap} = \text{pos}_n - \text{pos}_{n-1} - 1$
- C.f., Jiangong Zhang, Xiaohui Long and Torsten Suel: Performance of Compressed Inverted List Caching in Search Engines. WWW 2008.

# 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
- However, 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