

CS3245

Information Retrieval

Lecture 6: Index Compression

6



Live Q&A
<https://pollev.com/jin>

Last Time: index construction

- Sort-based indexing
 - 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 as they occur
- Distributed indexing using MapReduce
- Dynamic indexing: Multiple indices, logarithmic merge

Why compression?



- Use less disk space
- Keep more data (e.g., **the dictionary**) in memory
- Increase the speed of data (e.g., **the posting lists**) transfer from disk to memory

Today: Idx Cmprssn



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

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

CALPURNIA	→	2	31	54	101
-----------	---	---	----	----	-----

- Empirical laws on collection statistics (with RCV1)
- Dictionary compression
- Postings file compression



Reuters RCV1 statistics

symbol	statistic	value
N	documents	800,000
L	avg. # tokens per doc	200
M	terms	400,000
	(= vocabulary size = # of entries in the dictionary)	
	avg. # bytes per term	7.5
T	term-docID pairs	100,000,000
	(= tokens)	

Where do all those extra terms come from if English vocabulary is only ~30K?

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 $b \approx 0.5$
- An empirical finding ("empirical law")
- In a log-log plot of vocabulary size M vs. T , Heaps' law predicts a line with slope about $\frac{1}{2}$

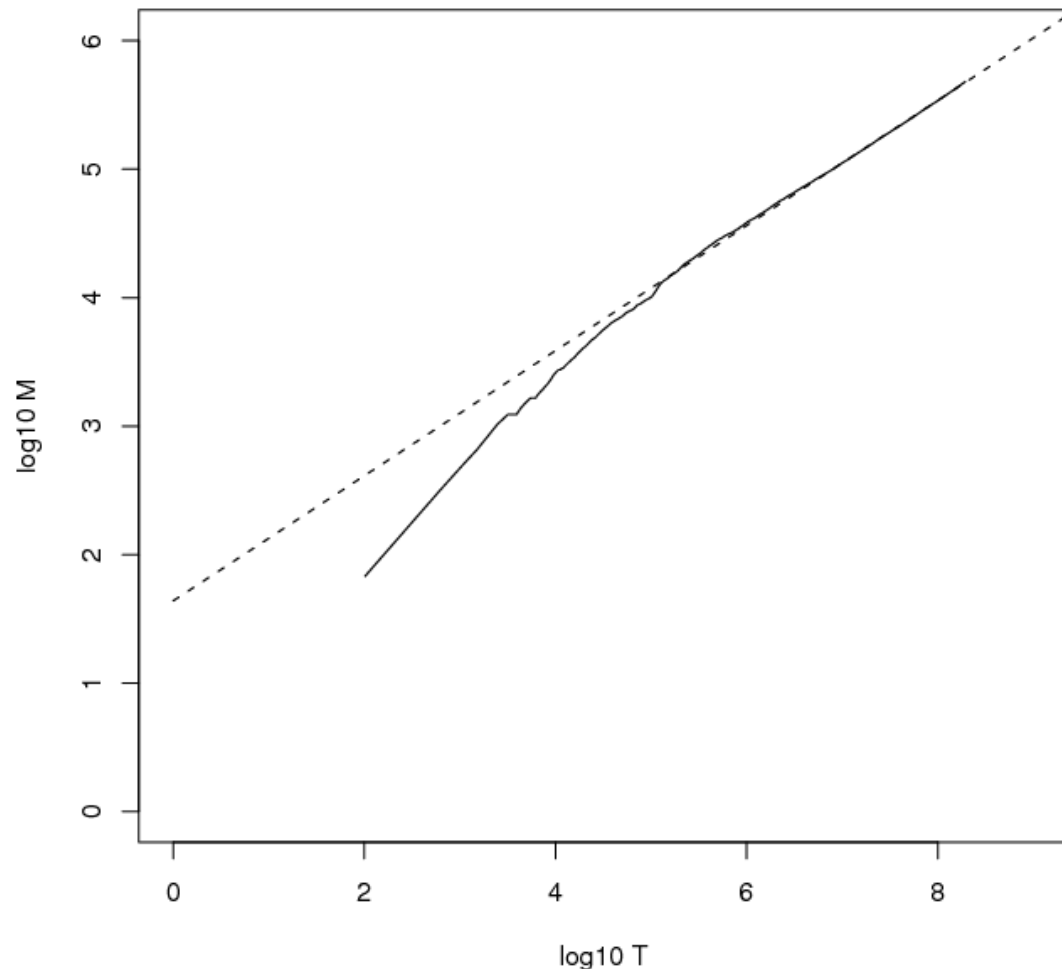
Heaps' Law

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



Collection frequency



- Some terms are common and some others are rare...
- Collection frequency (cf)
 - The number of occurrences of a term in the **collection**.
 - NOT the same as **document frequency** (df)
- Example
 - Collection D_1 : a a a b and D_2 : a b c
 - $cf_a = 4$, $df_a = 2$.
- Nevertheless, cf is positively correlated with df in general.

Zipf's law



$$cf_i = K/i$$

- cf_i is the cf of the i -th most frequency term
- K is a normalizing constant, $cf_1 = K / 1 = K$
- Example:
 - Collection D_1 : a a a b and D_2 : a b c
 - Estimated collection frequency (with $cf_1 = K = 4$):
 - For a, the 1st most frequent term, $cf_1 = K / 1 = 4$
 - For b, the 2nd most frequent term, $cf_2 = K / 2 = 2$
 - For c, the 3rd most frequent term, $cf_3 = K / 3 = 1.33$

Zipf's law



- 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: $\log cf_i = \log K - \log i$
 - Linear relationship between $\log cf_i$ and $\log i$
 - Another power law relationship

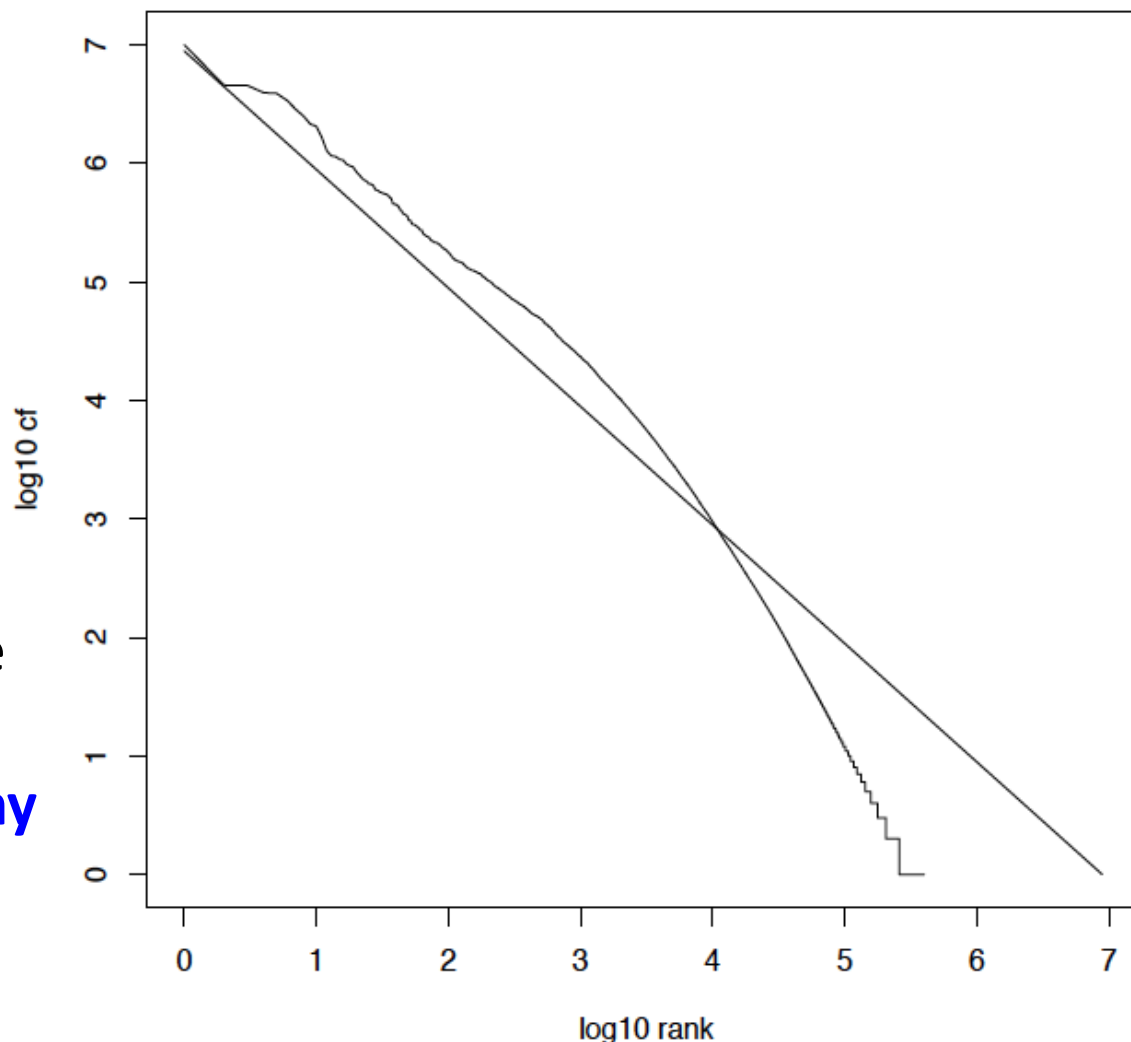
Zipf's law



Not a particularly good fit for RCV1...

But good enough as a rough model for calculations.

In general, there are **a few very frequent terms** and **very many very rare terms**.





DICTIONARY COMPRESSION



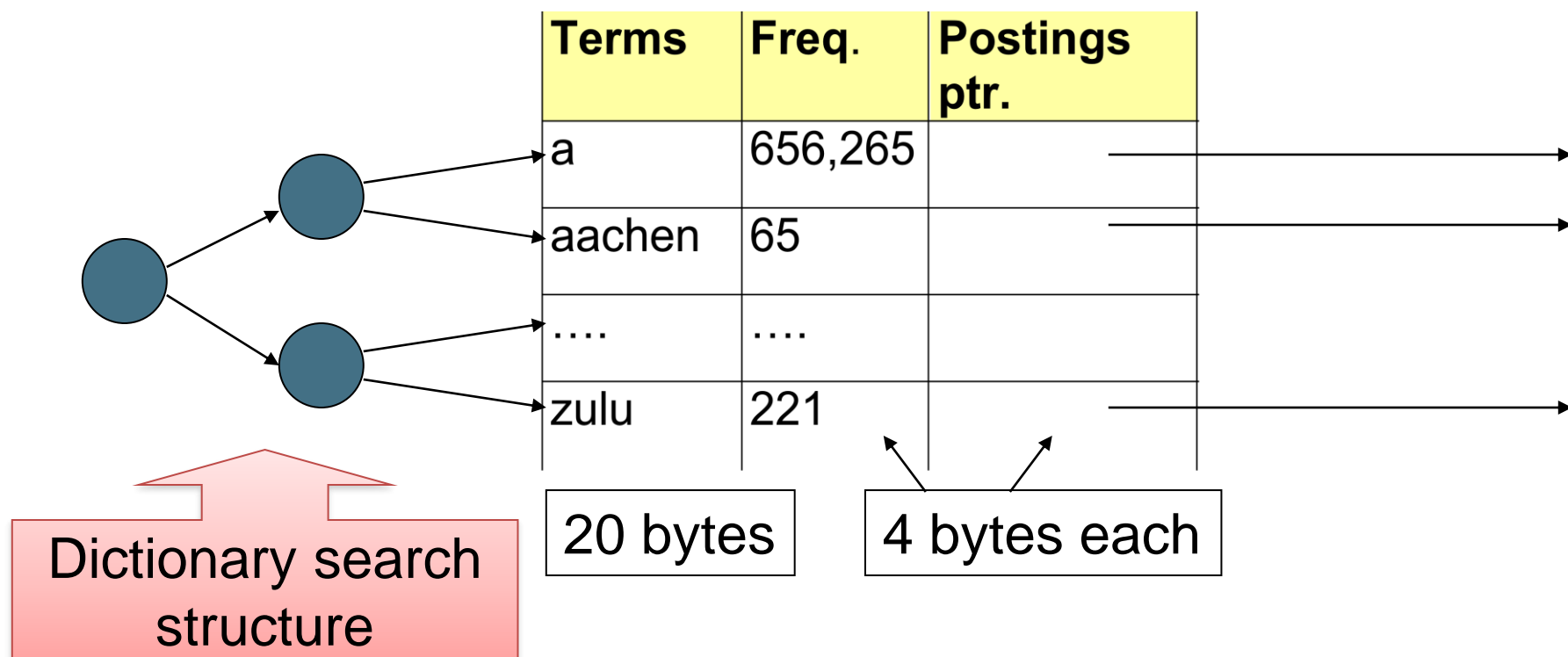
Why compress the dictionary?

- Search begins with the dictionary so 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

Compressing the dictionary is important

Dictionary storage - first cut

- Fixed-width entries indexed by a tree
 - ~400,000 terms; 28 bytes/term = 11.2 MB.





Fixed-width terms are wasteful

- Most of the bytes in the **Term** column are wasted
 - Average dictionary word in English: ~8 characters
 - And we still can't handle *supercalifragilisticexpialidocious* or *hydrochlorofluorocarbons*.

- How to save space?

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



- Store dictionary as a (long) string of characters
- Add pointers to the start of every word

....systileszygeticszygialszygyszaibelyiteszczeci....

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

Total string length =
400K × 8B = 3.2MB

Pointers resolve 3.2M
positions: $\log_2 3.2M =$
22bits = 3bytes



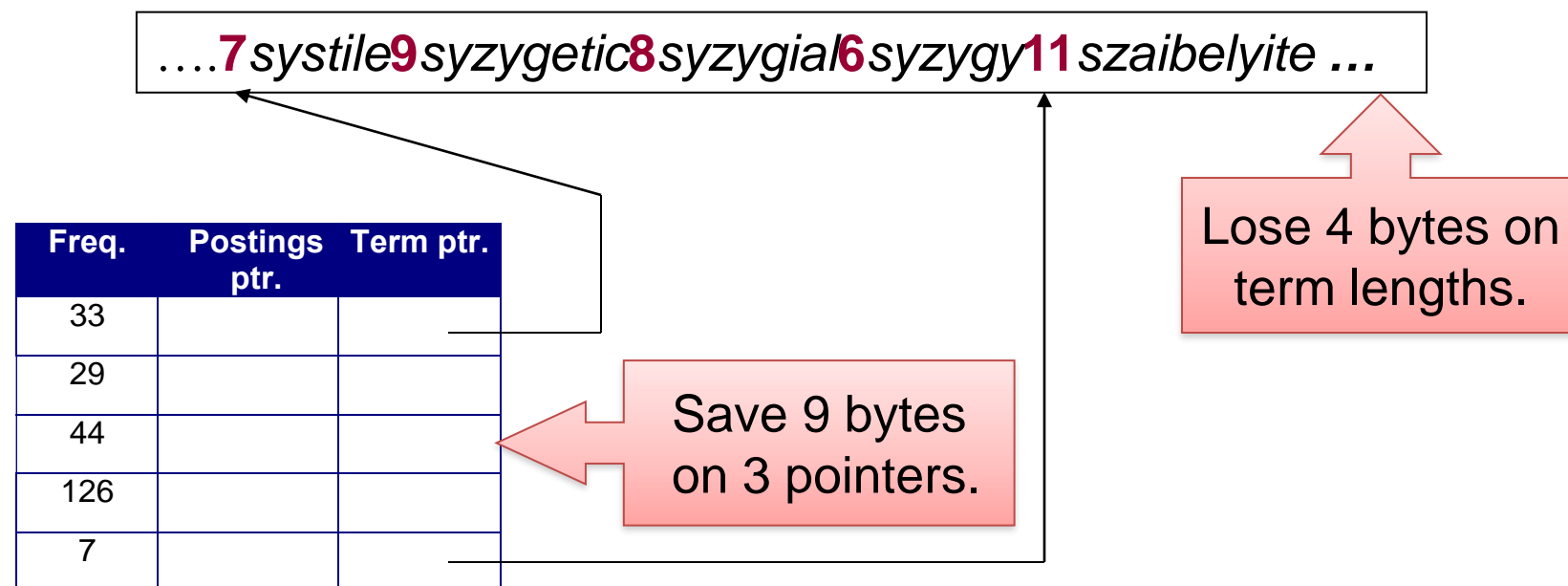
Space for dictionary as a string

- Dictionary array of 400K terms of 11 bytes each
 - 4 bytes per term for frequency
 - 4 bytes per term for pointer to postings
 - 3 bytes per term pointer

} Now avg. 11 bytes/term, not 28.
- Dictionary string of 400K terms of 8 bytes on average
- Total size = 4.4 MB (dictionary array)
+ 3.2 MB (dictionary string)
= 7.6 MB (3.6 MB less than the original size of 11.2MB)

Blocking

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





Net Result

- 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.

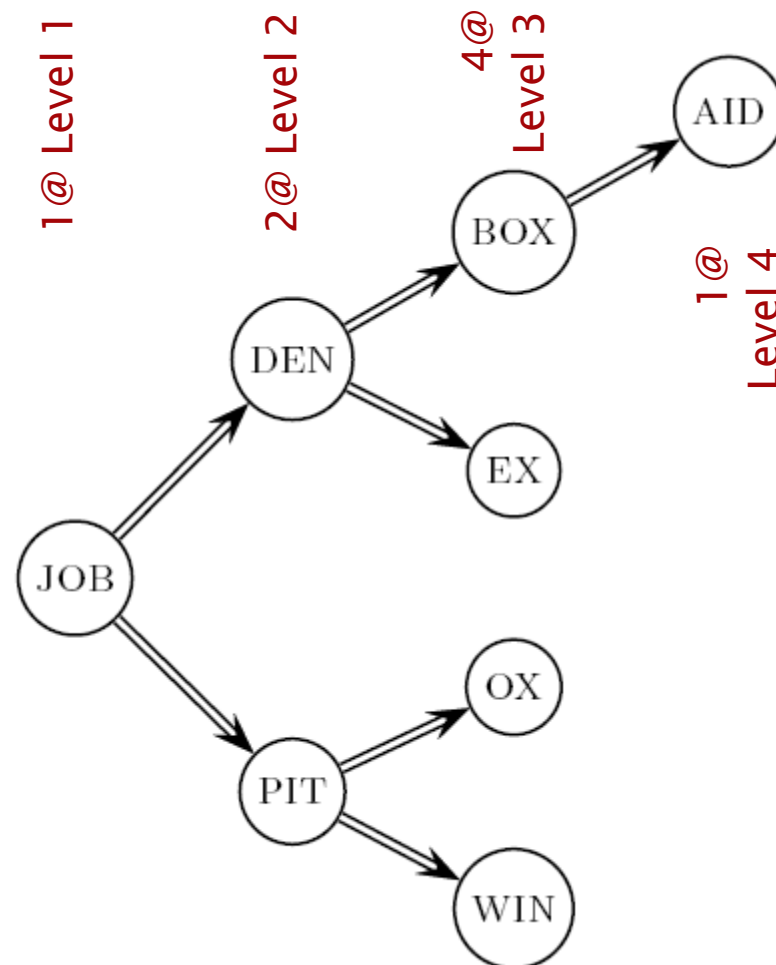
Shaved another ~ 0.5 MB. This reduces the size of the dictionary from 7.6 MB to 7.1 MB.

We can save more with larger k .

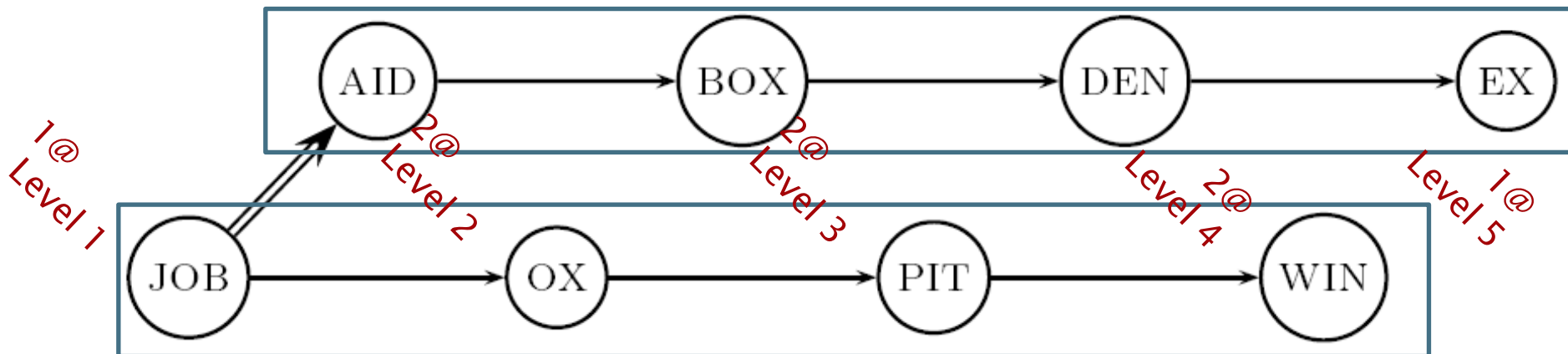
Why not go with a larger k ?

Dictionary search without blocking

- Assume that each dictionary term equally likely in query (not true in practice!)
- Average number of comparisons = $(1*1 + 2*2 + 3*4 + 4*1)/8$
= ~ 2.6



Dictionary search with blocking



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

Front coding

- Sorted words commonly have long common prefix – store differences only
 - Used for last $k-1$ terms in a block of k

8automata8automate9automatic10automation

→ **8automat* a1◇e2◇ic3◇ion**

Encodes **automat**

Extra length
beyond **automat.**

Begins to resemble general string compression

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 FILE COMPRESSION

Postings file compression



- How to store postings (i.e., **docIDs**) compactly?
 - *Computer, 34592: 33,47,154,159,202 ...*
- For Reuters (800,000 documents)
 - Range of docIDs [1, 800,000]
 - $\log_2 800000 \approx 20$ bits ≈ 3 bytes
- Let's try to make the numbers smaller!



Gap Encoding

- We store the list of docs containing a term in increasing order of docID.
 - *Computer, 34592*: 33,47,154,159,202 ...
- Consequence: it suffices to store *gaps*.
 - 33,14,107,5,43 ...

Gap Encoding



- As described by Zip's law, a small number of terms have a high cf and a lot of more words have a much lower cf .
- A high cf usually implies a high df , assuming the terms are evenly distributed across the documents.
- The gaps between the postings for a high df should be small.

Gap Encoding



	Encoding	Postings List					
the	docIDs	...	283042	283043	283044	283045	...
	gaps		...	1	1	1	
computer	docIDs	...	2803047	283154	283159	283202	...
	gaps		...	107	5	43	
arachno- centric	docIDs	252000	500100				
	gaps	...	248100				

Variable byte encoding



- Observation: it is wasteful and to use a fixed number of bits to store every number.
- Key challenge: encode every integer (gap) with about as little space as needed for that integer.
- This can be achieved by *variable byte encoding*, which uses *close to the fewest bytes* needed to store a gap.

Variable byte encoding



- Begin with one byte to store a gap G and dedicate 1 bit in it to be a continuation bit c
 - 0 (not ending) and 1 (ending)
- 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

824 = 1100111000 (binary)

5 = 101 (binary)

Postings stored as the byte concatenation

00000110 10111000 10000101 00001101 00001100 10110001

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 blend of variable-length coding and sensitivity to computer memory alignment

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 bits)	400.0
postings, uncompressed (20 bits)	250.0
postings, variable byte encoded	116.0



Summary: Index compression

- We can now create an index for highly efficient Boolean retrieval that is very space efficient
- Use the sorted nature of the data to compress
 - Variable sized storage
 - Encode common prefixes only once
 - Encode gaps to reduce size of numbers
- However, here we didn't encode positional information
 - But techniques for dealing with postings are similar



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