Index Compression

David Kauchak
cs160
Fall 2009

adapted from:
http://www.stanford.edu/class/cs276/handouts/lecture5-indexcompression.ppt

Administrative

- Homework 2
- Assignment 1
- Assignment 2
  - Pair programming?

Topic Detection and Tracking

Topic Detection and Tracking (TDT) is a multi-site research project, now in its third phase, to develop core technologies for a news understanding systems. Specifically, TDT systems discover the topical structure in unsegmented streams of news reporting as it appears across multiple media and in different languages. For a detailed discussion of the goals of TDT, see Charles Wayne's overview. The NIST web site describes the evaluation methodology and reports on previous phases of TDT research. LDC developed the corpus for the second phase of TDT and is currently developing the phase three corpus. More detailed information follows on the phases of TDT and the corpora they involve.

- Pilot-Study
- TDT 2 *(corpus used for training and for 1998 test)*
- TDT 3 *(corpus used in 1999, 2000 and 2001 tests)*
- TDT 2000 -- takes you to the NIST TDT-2000 web page
- TDT 2001 -- takes you to the NIST TDT-2001 web page
- TDT 4 *(corpus used for 2002, 2003 tests)*
- TDT 5 *(corpus used for 2004 test)*

RCV1 token normalization

size of	word types (terms)		
	dictionary		
	Size (K)	∆%	cumul %
Unfiltered	484		
No numbers	474	-2	-2
Case folding	392	-17	-19
30 stopwords	391	-0	-19
150 stopwords	391	-0	-19
stemming	322	-17	-33

TDT token normalization

normalization	terms	% change
none	120K	-
number folding	117K	3%
lowercasing	100K	17%
stemming	95K	25%
stoplist	120K	0%
number & lower & stoplist	97K	20%
all	78K	35%

What normalization technique(s) should we use?

Index parameters vs. what we index

size of	word types (terms)			non-positional postings			positional postings		
	dictionary			non-positional index			positional index		
	Size (K)	∆%	cumul %	Size (K)	∆ %	cumul %	Size (K)	∆ %	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

Index parameters vs. what we index

size of	word types (terms)			non-positional postings			positional postings		
	dictionary			non-positional index			positional index		
	Size (K)	∆%	cumul %	Size (K)	∆ %	cumul %	Size (K)	∆ %	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

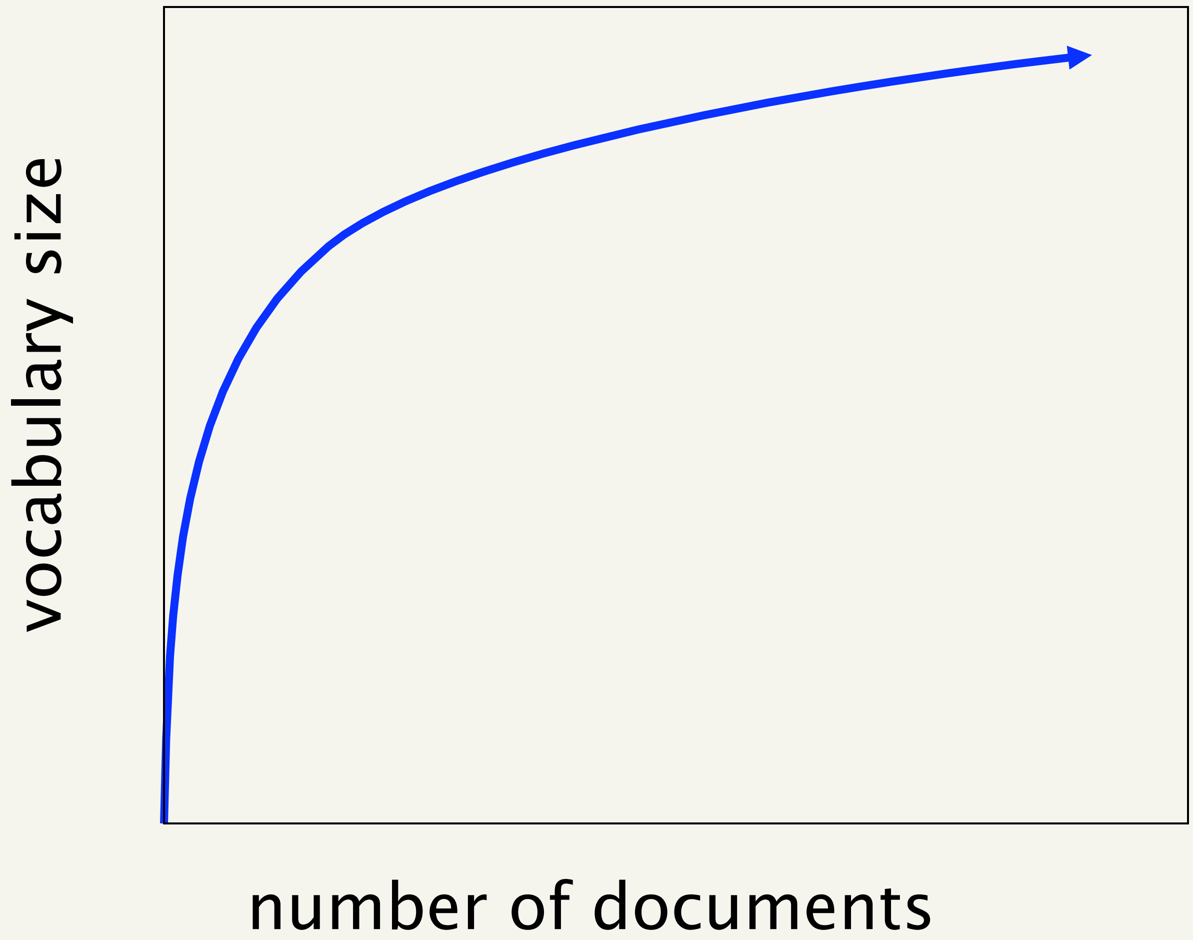
Index parameters vs. what we index

size of	word types (terms)			non-positional postings			positional postings		
	dictionary			non-positional index			positional index		
	Size (K)	∆%	cumul %	Size (K)	∆ %	cumul %	Size (K)	∆ %	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

Corpora statistics

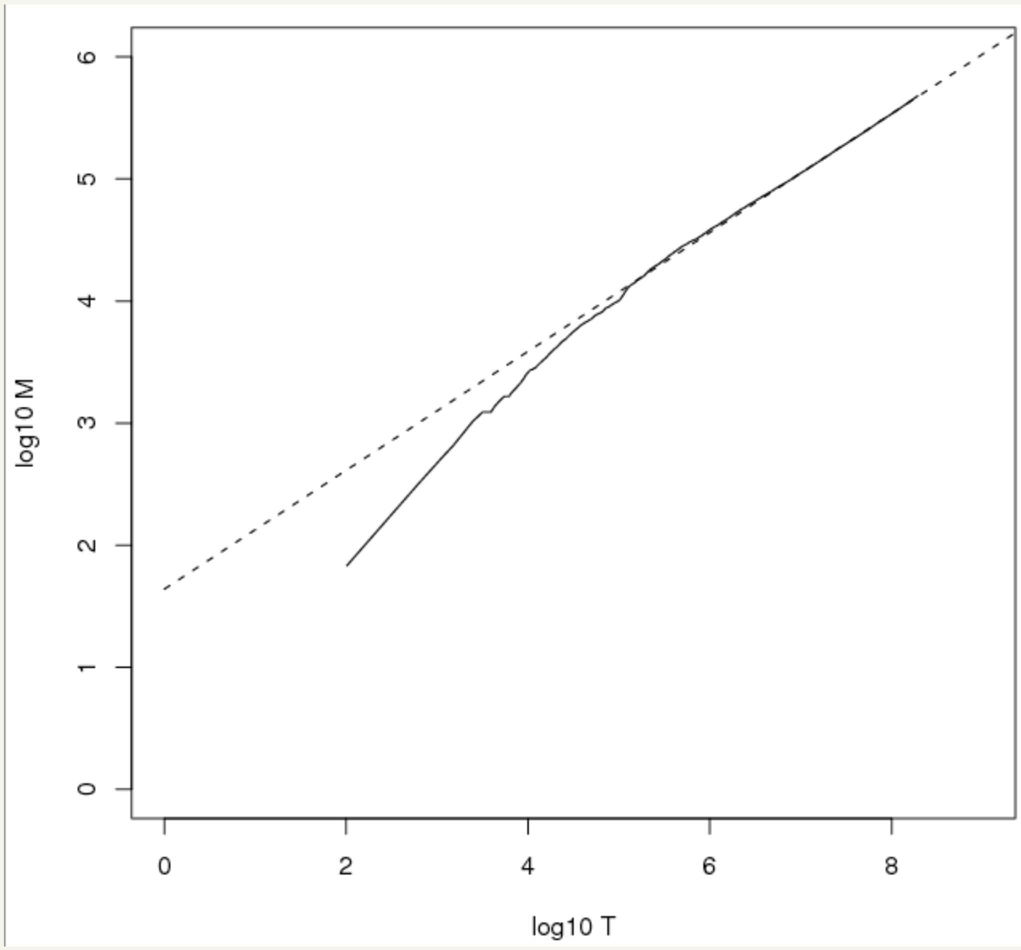
statistic	TDT	Reuters RCV1
documents	16K	800K
avg. # of tokens per doc	400	200
terms	100K	400K
non-positional postings	?	100M

How does the vocabulary size grow with the size of the corpus?



How does the vocabulary size grow with the size of the corpus?

log of the vocabulary size



log of the number of documents

Heaps' law

$$\text{Vocab size} = k\ (\text{tokens})^b$$

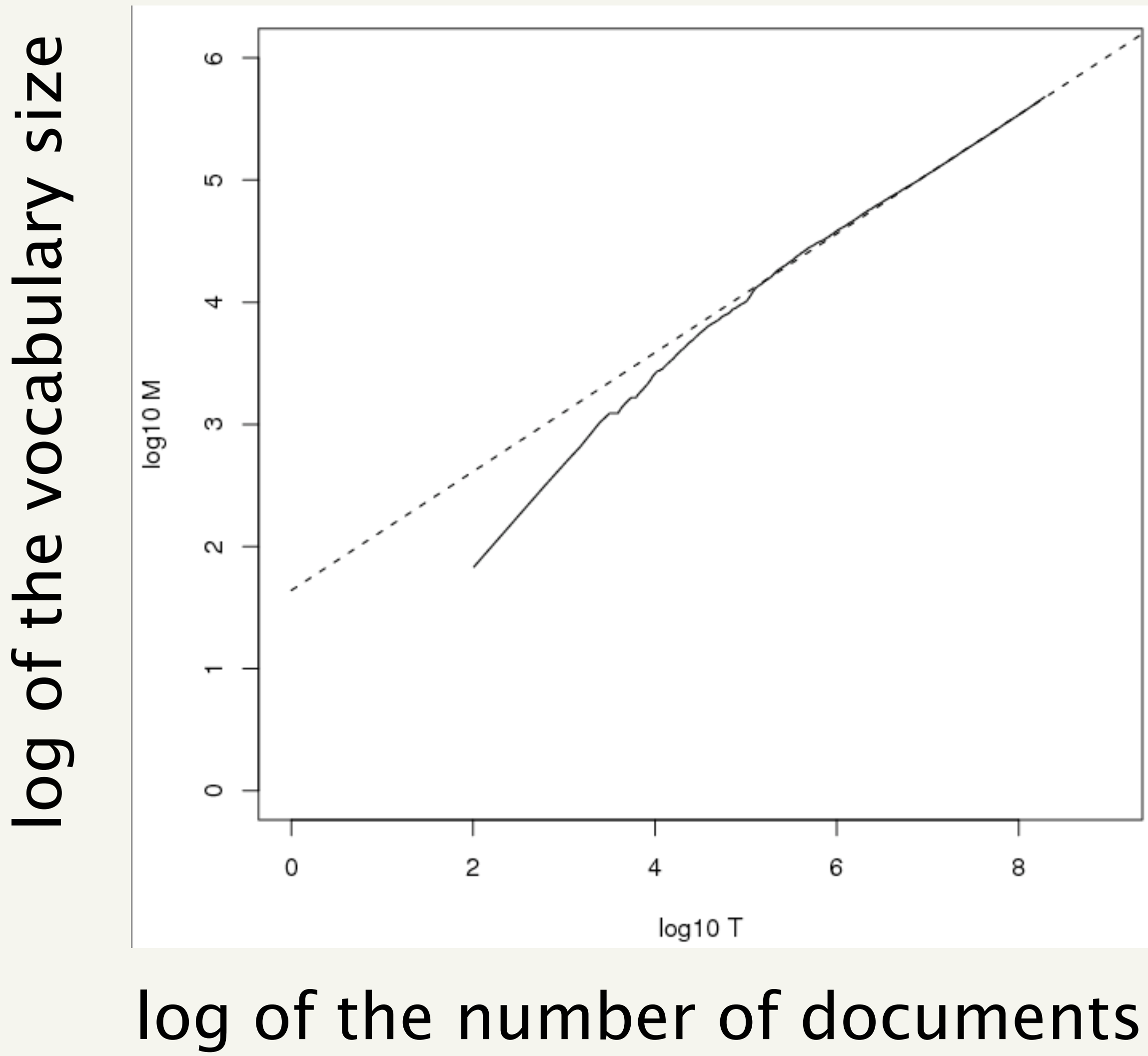
$$M = k\, T^b$$

- Typical values: $30 \le k \le 100$ and $b \approx 0.5$.
- Does this explain the plot we saw before?

$$\log M = \log k + b \log(T)$$

- What does this say about the vocabulary size as we increase the number of documents?
  - there are almost always new words to be seen: increasing the number of documents increases the vocabulary size
  - to get a linear increase in vocab size, need to add exponential number of documents

How does the vocabulary size grow with the size of the corpus?



$\log_{10} M = 0.49 \log_{10} T + 1.64$ is the best least squares fit.

$M = 10^{1.64} T^{0.49}$

$k = 10^{1.64} \approx 44$

$b = 0.49$.

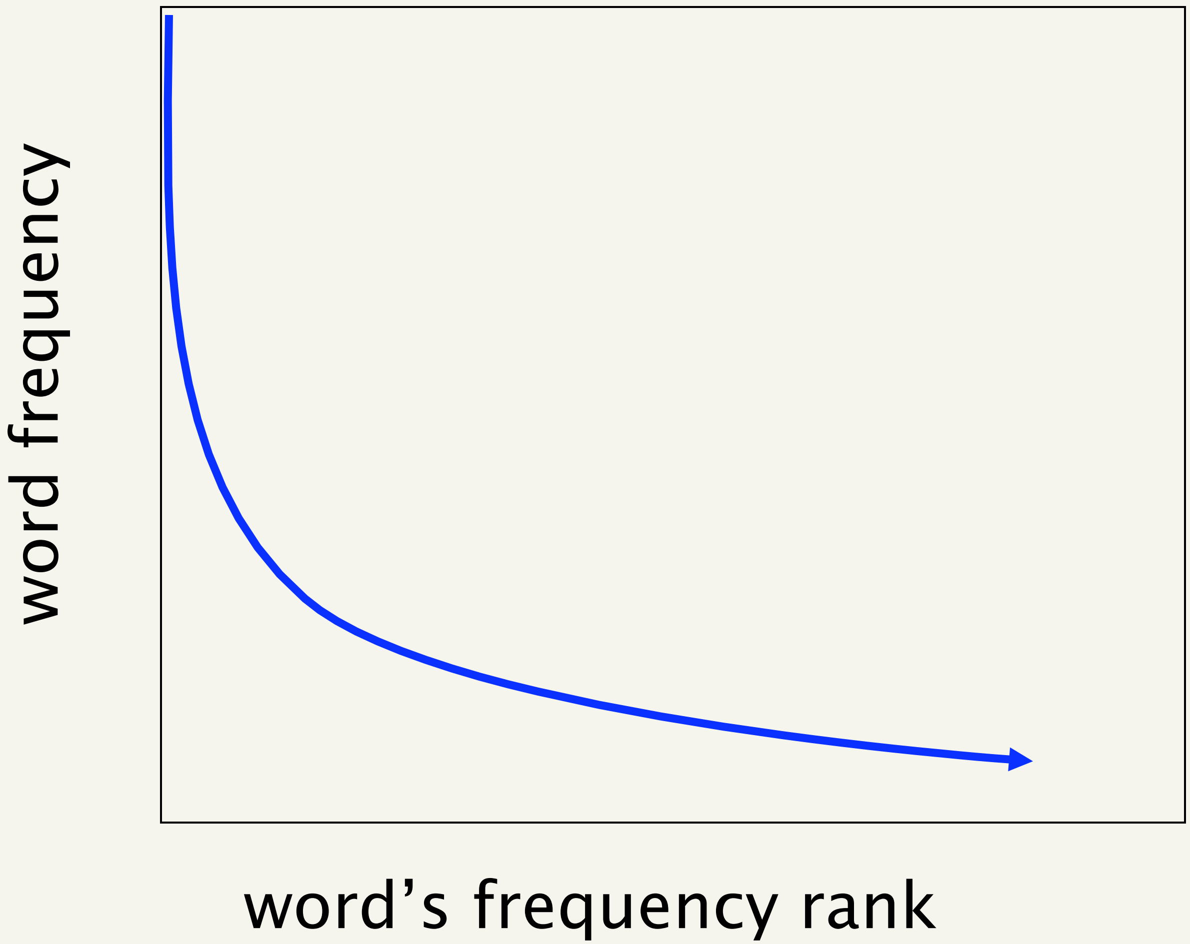
Discussion

- How do token normalization techniques and similar efforts like spelling correction interact with Heaps' law?

Heaps' law and compression

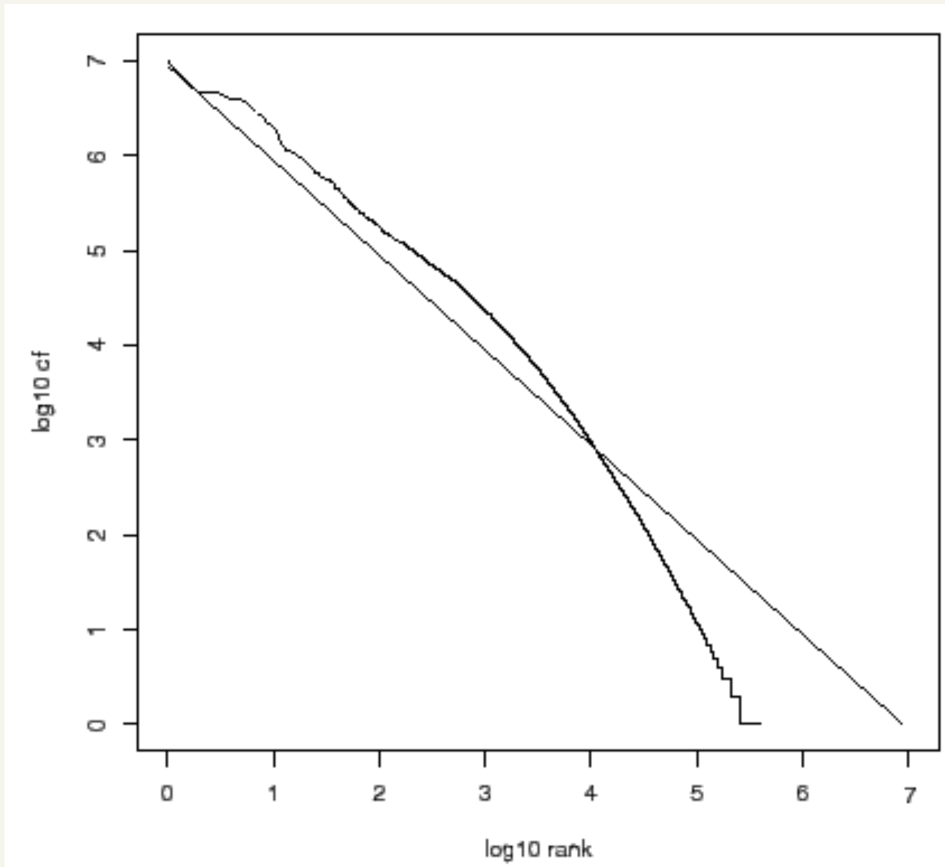
- Today, we're talking about index compression, i.e. reducing the memory requirement for storing the index
- What implications does Heaps' law have for compression?
  - Dictionary sizes will continue to increase
  - Dictionaries can be very large

How does a word's frequency relate to it's frequency rank?



How does a word's frequency relate to it's frequency rank?

log of the frequency



log of the frequency rank

Zipf's law

- 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*

$$\text{frequency}_i \propto c/i$$

- where *c* is a constant

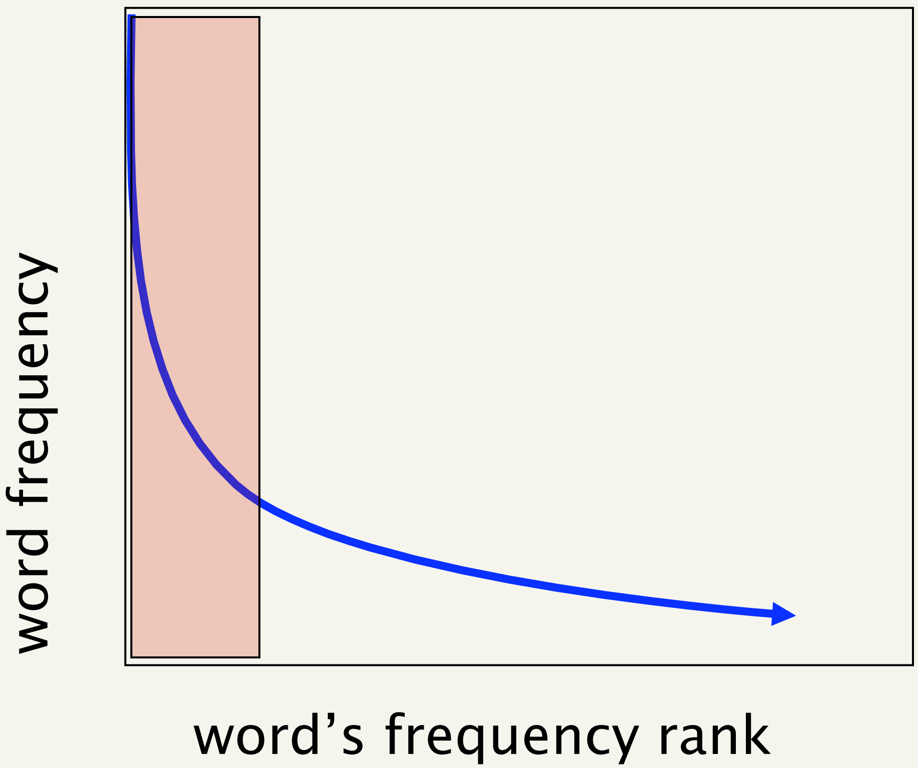
$$\log(\text{frequency}_i) \propto \log c - \log i$$

Consequences of Zipf's law

- If the most frequent term (*the*) occurs cf_1 times, how often do the 2nd and 3rd most frequent occur?
  - then the second most frequent term (*of*) occurs $cf_1/2$ times
  - the third most frequent term (*and*) occurs $cf_1/3$ times …
- If we're counting the number of words in a given frequency range, lowering the frequency band linearly results in an exponential increase in the number of words

Zipf's law and compression

- What implications does Zipf's law have for compression?



Some terms will occur **very** frequently in positional postings lists

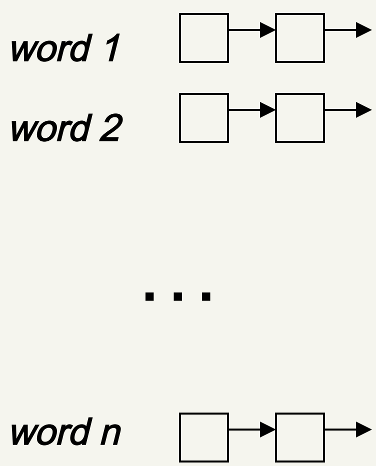
Dealing with these well can drastically reduce the index size

Index compression

- Compression techniques attempt to decrease the space required to store an index

- What other benefits does compression have?
  - Keep more stuff in memory (increases speed)
  - Increase data transfer from disk to memory
    - [read compressed data and decompress] is faster than [read uncompressed data]
    - What does this assume?
      - Decompression algorithms are fast
      - True of the decompression algorithms we use

Inverted index



What do we need to store?

How are we storing it?

Compression in inverted indexes

- First, we will consider space for dictionary
  - Make it small enough to keep in main memory
- Then the postings
  - Reduce disk space needed, decrease time to read from disk
  - Large search engines keep a significant part of postings in memory

Lossless vs. lossy compression

- What is the difference between lossy and lossless compression techniques?
- Lossless compression: All information is preserved
- Lossy compression: Discard some information, but attempt to keep information that is relevant
  - Several of the preprocessing steps can be viewed as lossy compression: case folding, stop words, stemming, number elimination.
  - Prune postings entries that are unlikely to turn up in the top *k* list for any query
- Where else have you seen lossy and lossless compresion techniques?

Why compress the dictionary

- Must keep in memory
  - Search begins with the dictionary
  - Memory footprint competition
  - Embedded/mobile devices

What is a straightforward way of storing the dictionary?

What is a straightforward way of storing the dictionary?

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

Terms	**Freq.**	**Postings ptr.**
a	656,265	
aachen	65	
….	….	
zulu	221	

20 bytes

4 bytes each

Fixed-width terms are wasteful

- Any problem with this approach?
  - 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
- Written English averages ~4.5 characters/word
  - Is this the number to use for estimating the dictionary size?
- Ave. dictionary word in English: ~8 characters
- Short words dominate token counts but not type average

Any ideas?

- Store the dictionary as one long string

….systilesyzygeticsyzygialsyzygyszaibelyiteszczecinszomo….

- Gets ride of wasted space
- If the average word is 8 characters, what is our savings over the 20 byte representation?
- Theoretically, 60%
- Any issues?

Dictionary-as-a-String

- Store dictionary as a (long) string of characters:
  - Pointer to next word shows end of current word

….systilesyzygeticsyzygialsyzygyszaibelyiteszczecinszomo….

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

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

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

How much memory to store the pointers?

Space for dictionary as a string

- Fixed-width
  - 20 bytes per term = 8 MB
- As a string
  - 6.4 MB (3.2 for dictionary and 3.2 for pointers)
- 20% reduction!

- Still a long way from 60%. Any way we can store less pointers?

Blocking

- Store pointers to every *k*th term string

….systilesyzygeticsyzygialsyzygyszaibelyiteszczecinszomo….

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

What else do we need?

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

- Where we used 3 bytes/pointer without blocking
  - 3 x 4 = 12 bytes for k=4 pointers,

now we use 3+4=7 bytes for 4 pointers.

Shaved another ~0.5MB; can save more with larger k.

Why not go with larger k?

Dictionary search without blocking

- How would we search for a dictionary entry?

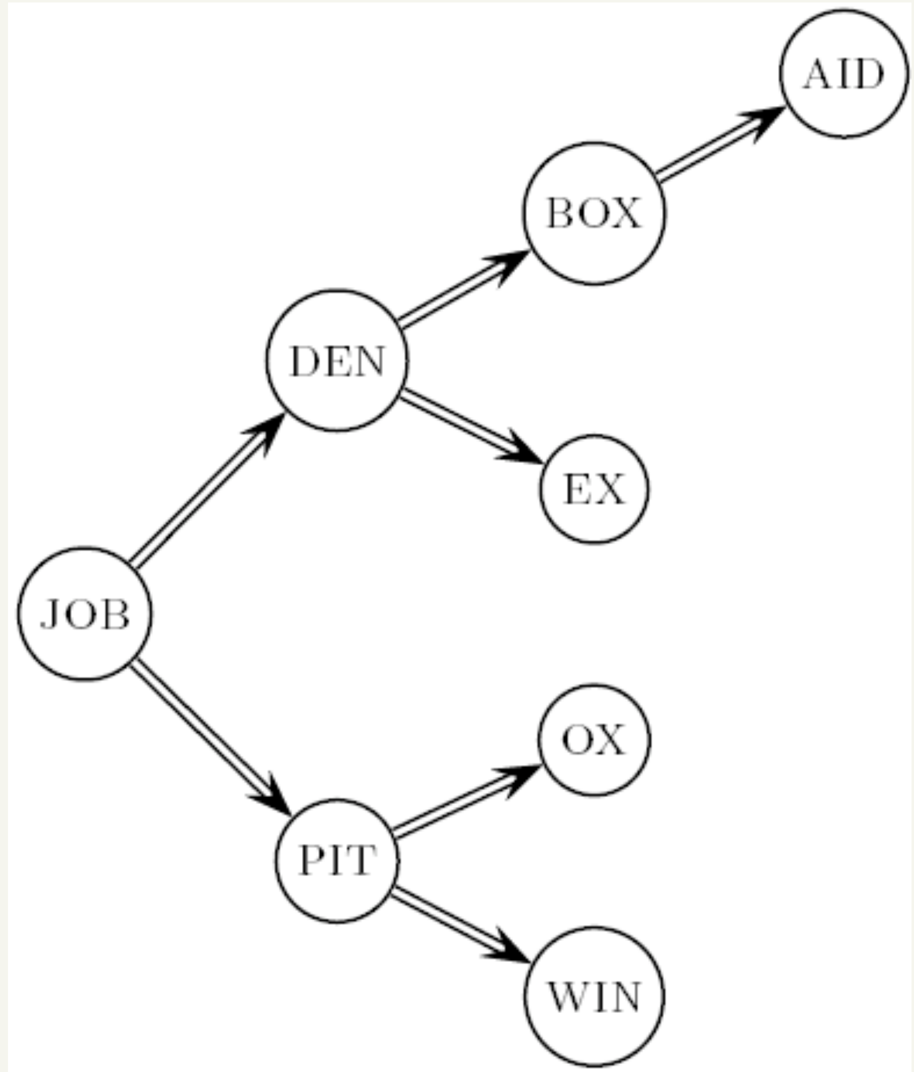
….systilesyzygeticsyzygialsyzygyszaibelyiteszczecinszomo….

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

Dictionary search without blocking

- Binary search

- Assuming each dictionary term is equally likely in query (not really so in practice!), average number of comparisons = ?
- (1+2·2+4·3+4)/8 ~2.6



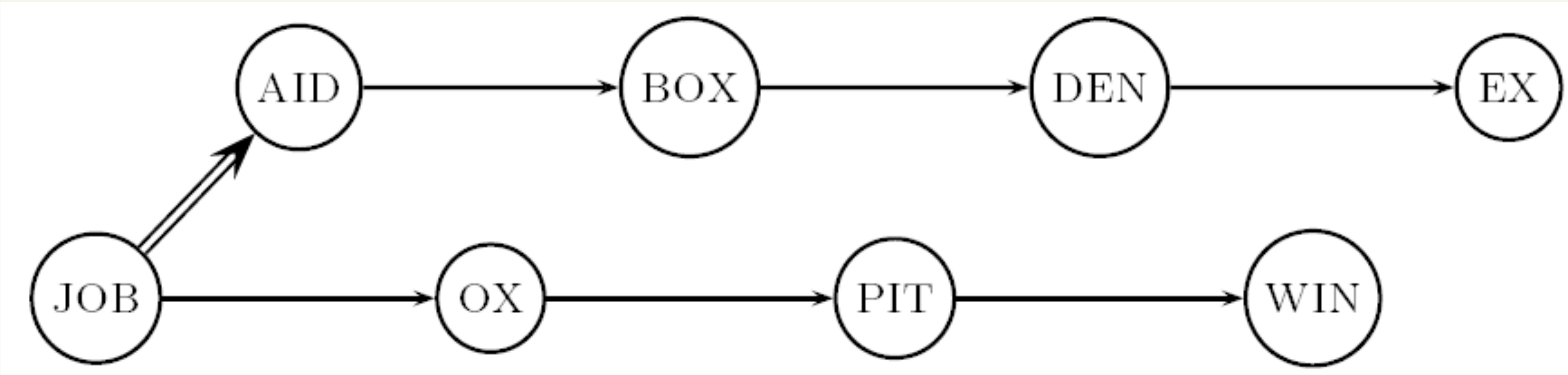
Dictionary search with blocking

- What about with blocking?

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

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

Dictionary search with blocking



- Binary search down to 4-term block
  - Then linear search through terms in block.
- Blocks of 4 (binary tree), avg. = ?
- $(1+2\cdot2+2\cdot3+2\cdot4+5)/8 = 3$ compares

More improvements…

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

- We’re storing the words in sorted order
- Any way that we could further compress this block?

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****a1*e*2*ic*3*ion***

Encodes ***automat***

Extra length beyond ***automat***

Begins to resemble general string compression

RCV1 dictionary compression

Technique	Size in MB
Fixed width	11.2
String with pointers to every term	7.6
Blocking *k* = 4	7.1
Blocking + front coding	5.9

Postings compression

- The postings file is much larger than the dictionary, by a factor of at least 10

- A posting for our purposes is a docID
- Regardless of our postings list data structure, we need to store all of the docIDs

- 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

Postings: two conflicting forces

- Where is most of the storage going?
- Frequent terms will occur in most of the documents and require a lot of space
- A term like ***the*** occurs in virtually every doc, so 20 bits/posting is too expensive.
  - Prefer 0/1 bitmap vector in this case
- 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.

Postings file entry

- We store the list of docs containing a term in increasing order of docID.
  - ***computer***: 33,47,154,159,202 …

- Is there another way we could store this sorted data?
- Store *gaps*: 33,14,107,5,43 …
  - 14 = 47-33
  - 107 = 154 – 47
  - 5 = 159 - 154

Fixed-width

	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								

- How many bits do we need to encode the gaps?
- Does this buy us anything?

Variable length encoding

- Aim:
  - For ***arachnocentric***, we will use ~20 bits/gap entry
  - For ***the***, we will use ~1 bit/gap entry
- Key challenge: encode every integer (gap) with as few bits as needed for that integer

1, 5, 5000, 1, 1524723, …

for smaller integers, use fewer bits
for larger integers, use more bits

Variable length coding

1, 5, 5000, 1, 1124 …

1, 101, 1001110001, 1, 10001100101 …

Fixed width:

0000000001000000010110011100001 …

every 10 bits

Variable width:

1101100111000111000110 0101 …

?

Variable Byte (VB) codes

- Rather than use 20 bits, i.e. record gaps with the smallest number of bytes to store the gap

1, 101, 1001110001

$\underbrace{00000001}_{\text{1 byte}}, \underbrace{00000101}_{\text{1 byte}}, \underbrace{00000010\ 01110001}_{\text{2 bytes}}$

000000010000010100000010011110001

?

VB codes

- Reserve the first bit of each byte as the continuation bit
- If the bit is 1, then we're at the end of the bytes for the gap
- If the bit is 0, there are more bytes to read

1, 101, 1001110001

10000001100001010000100 11110001

- For each byte used, how many bits of the gap are we storing?

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.

Other variable codes

- Instead of bytes, we can also use a different “unit of alignment”: 32 bits (words), 16 bits, 4 bits (nibbles) etc.
- What are the pros/cons of a smaller/larger unit of alignment?
  - Larger units waste less space on continuation bits (1 of 32 vs. 1 of 8)
  - Smaller unites waste less space on encoding smaller number, e.g. to encode ‘1’ we waste (6 bits vs. 30 bits)

More codes

1000000011000001010000000100 11110001

- Still seems wasteful
- What is the major challenge for these variable length codes?
- We need to know the length of the number!

- Idea: Encode the length of the number so that we know how many bits to read

Gamma codes

- Represent a gap as a pair *length* and *offset*
- *offset* is *G* in binary, with the leading bit cut off
  - 13 → 1101 → 101
  - 17 → 10001 → 0001
  - 50 → 110010 → 10010
- *length* is the length of offset
  - 13 (offset 101), it is 3
  - 17 (offset 0001), it is 4
  - 50 (offset 10010), it is 5

Encoding the length

- We've stated *what* the length is, but not *how* to encode it
- What is a requirement of our length encoding?
  - Lengths will have variable length (e.g. 3, 4, 5 bits)
  - We must be able to decode it without any ambiguity
- Any ideas?
- Unary code
  - Encode a number *n* as *n* 1's, followed by a 0, to mark the end of it
  - 5 → 111110
  - 12 → 1111111111110

Gamma code examples

number	length	offset	γ-code
0			
1			
2			
3			
4			
9			
13			
24			
511			
1025			

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

Gamma code properties

- Uniquely prefix-decodable, like VB
- All gamma codes have an odd number of bits

- What is the fewest number of bits we could expect to express a gap (without any other knowledge of the other gaps)?
  - $\log_2(\text{gap})$
- How many bits do gamma codes use?
  - $2\lfloor \log_2(\text{gap}) \rfloor + 1$ bits
  - Almost within a factor of 2 of best possible

Gamma seldom used in practice

- Machines have word boundaries – 8, 16, 32 bits
- Compressing and manipulating at individual bit-granularity will slow down query processing
- Variable byte alignment is potentially more efficient
- Regardless of efficiency, variable byte is conceptually simpler at little additional space cost

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

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 <u>text</u> 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

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