605.744 Information Retrieval

Efficiency Issues

JOHNS HOPKINS
WHITING SCHOOL
of ENGINEERING

Module 3 Overview

- Review Inverted File Construction

- Chapter 5: Index Compression

  Zipf’s Law & Heap’s Law (IIR 5.1)

  Dictionaries (IIR 5.2)

  Inverted files (IIR 5.3)

Recap: Indexing

- What will the dictionary / inverted file look like for these documents (use Algorithm A)

  Assume all integers can be stored in 1 byte

  Write out inverted file to disk in order A to Z

  ○ all boy cows deserves eat every fudge good grass

- Doc 1: every good boy deserves fudge
- Doc 2: all cows eat all grass
- Doc 3: good boy deserves fudge
- Doc 4: good boy deserves all fudge

Inverted File Exercise: Solution

Doc 1: every good boy deserves fudge
Doc 2: all cows eat all grass
Doc 3: good boy deserves fudge
Doc 4: good boy deserves all fudge

every	(1,1)		
good	(1,1)	(3,1)	(4,1)
boy	(1,1)	(3,1)	(4,1)
deserves	(1,1)	(3,1)	(4,1)
fudge	(1,1)	(3,1)	(4,1)
all	(2,2)	(4,1)	
cows	(2,1)		
eat	(2,1)		
grass	(2,1)		

(docid, term count)

Dictionary

Term	DF	Offset
all	2	0
boy	3	4
cows	1	10
deserves	3	12
eat	1	18
every	1	20
fudge	3	22
good	3	28
grass	1	34

Inverted File

2	2	4	1	1	1	3	1	4	1
2	1	1	1	3	1	4	1	2	1
1	1	1	1	3	1	4	1	1	1
3	1	4	1	2	1				

Algorithm E

- Acts like Alg A
  - Scan documents. Build lexicon and postings in memory.
- When memory is exhausted
  - Write a partial index out for the currently examined documents
  - Keep the to-date lexicon in memory, and flush all of the postings out to disk.
- Continue with the next set of documents
  - The lexicon grows (with new terms) and postings for the next traunche of documents accumulate until memory again fills up
- After seeing all documents
  - Write lexicon to disk
  - Perform an n-way merge on the partial indexes (the various bunches of postings)

Example w/ Alg. E (Docs 1&2)

Doc 1: every good boy deserves fudge
Doc 2: all cows eat all grass
Doc 3: good boy deserves fudge
Doc 4: good boy deserves all fudge

Dictionary

Term	ID	DF
all	0	1
boy	1	1
cows	2	1
deserves	3	1
eat	4	1
every	5	1
fudge	6	1
good	7	1
grass	8	1

Postings (Hashtable in memory)

every	(1,1)
good	(1,1)
boy	(1,1)
deserves	(1,1)
fudge	(1,1)
all	(2,2)
cows	(2,1)
eat	(2,1)
grass	(2,1)

(docid, term count)

IF #1:

0	2	2	1	1	1	2	2	1	3
1	1	4	2	1	5	1	1	6	1
1	7	1	1	8	2	1			

(termid, docid, term count)

Example w/ Alg. E (Docs 3&4)

Doc 1: every good boy deserves fudge
Doc 2: all cows eat all grass
Doc 3: good boy deserves fudge
Doc 4: good boy deserves all fudge

Dictionary

Term	ID	DF
all	0	2
boy	1	3
cows	2	1
deserves	3	3
eat	4	1
every	5	1
fudge	6	3
good	7	3
grass	8	1

Postings (Hashtable in memory)

good	(3,1)	(4,1)
boy	(3,1)	(4,1)
deserves	(3,1)	(4,1)
fudge	(3,1)	(4,1)
all	(4,1)	

(docid, term count)

IF #2:

0	4	1	1	3	1	1	4	1	3
3	1	3	4	1	6	3	1	6	4
1	7	3	1	7	4	1			

(termid, docid, term count)

N-way Merge

IF #1:

0	2	2	1	1	1	2	2	1	3
1	1	4	2	1	5	1	1	6	1
1	7	1	1	8	2	1			

IF #2:

0	4	1	1	3	1	1	4	1	3
3	1	3	4	1	6	3	1	6	4
1	7	3	1	7	4	1			

(termid, docid, term count)

Term	ID	DF	Offset
all	0	2	0
boy	1	3	4
cows	2	1	10
deserves	3	3	12
eat	4	1	18
every	5	1	20
fudge	6	3	22
good	7	3	28
grass	8	1	34

Final Inverted File

2	2	4	1	1	1	3	1	4	1
2	1	1	1	3	1	4	1	2	1
1	1	1	1	3	1	4	1	1	1
3	1	4	1	2	1				

Module 3 Overview

- Review Inverted File Construction

- Chapter 5: Index Compression

  Zipf’s Law & Heap’s Law (IIR 5.1)

  Dictionaries (IIR 5.2)

  Inverted files (IIR 5.3)

Corpus size for estimates

- Consider *N* = 1M documents, each with about *L*=1K terms.
- Avg 6 bytes/term incl. spaces/punctuation
  - 6GB of data.
- Say there are *m* = 1M *distinct* terms among these.

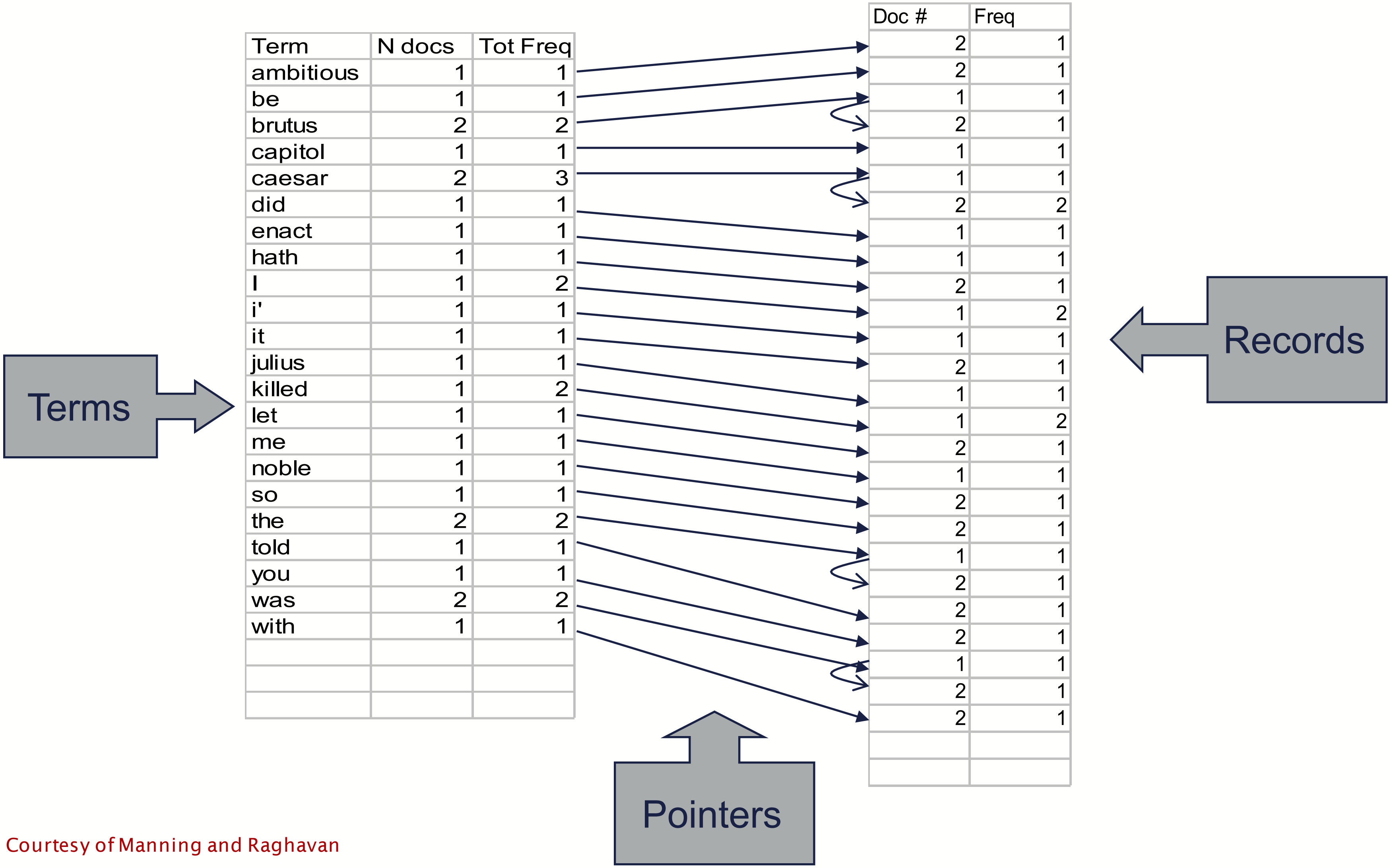
Courtesy of Manning and Raghavan

Recall: Don’t build the matrix

- A 1M x 1M matrix has a trillion 0’s and 1’s.
- But it has no more than one billion 1’s.
  - matrix is extremely sparse.
- So we devised the inverted index
  - Designed query processing to work on it
- What are the storage costs?

Courtesy of Manning and Raghavan

- Where do we pay in storage?



Term	N docs	Tot Freq
ambitious	1	1
be	1	1
brutus	2	2
capitol	1	1
caesar	2	3
did	1	1
enact	1	1
hath	1	1
I	1	2
i'	1	1
it	1	1
julius	1	1
killed	1	2
let	1	1
me	1	1
noble	1	1
so	1	1
the	2	2
told	1	1
you	1	1
was	2	2
with	1	1

Doc #	Freq
2	1
2	1
1	1
2	1
1	1
1	1
2	2
1	1
1	1
2	1
1	2
1	1
2	1
1	1
1	2
2	1
1	1
2	1
2	1
1	1
2	1
2	1
2	1
1	1
2	1
2	1

Courtesy of Manning and Raghavan

Index size

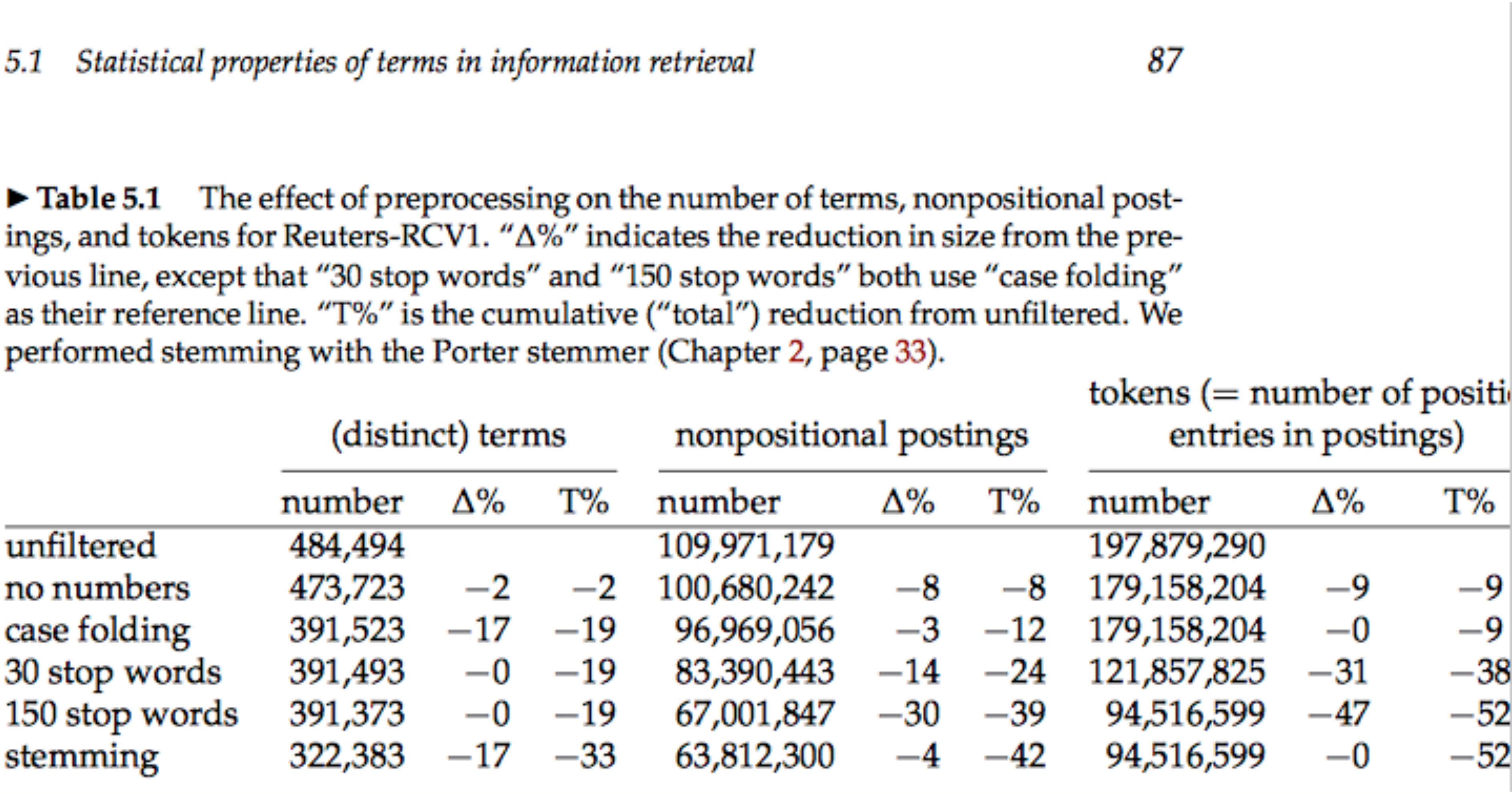
- Stemming/case folding/no numbers cuts
  - number of terms by ~35% [Paul says: up to 50%]
  - number of non-positional postings by 10-20%

- Stop words
  - Rule of 30: ~30 words account for ~30% of all term occurrences in written text [= # positional postings]
  - Eliminating 150 commonest terms from index will reduce non-positional postings ~30% *without considering compression*

  With compression, you save ~10% [Paul says: more like 5%]

Courtesy of Manning and Raghavan

Table 5.1



▶ **Table 5.1** The effect of preprocessing on the number of terms, nonpositional postings, and tokens for Reuters-RCV1. "Δ%" indicates the reduction in size from the previous line, except that "30 stop words" and "150 stop words" both use "case folding" as their reference line. "T%" is the cumulative ("total") reduction from unfiltered. We performed stemming with the Porter stemmer (Chapter 2, page 33).

	(distinct) terms			nonpositional postings			tokens (= number of positi entries in postings)		
	number	Δ%	T%	number	Δ%	T%	number	Δ%	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

Storage analysis

- First, we will consider space for postings
  - Basic Boolean index only
  - No analysis for positional indexes, etc.
  - Then we'll examine compression schemes

- And we'll do the same for the dictionary

Courtesy of Manning and Raghavan

Postings: two conflicting forces

- A term like ***Calpurnia*** occurs in maybe one doc out of a million – we would like to store this posting using $\log_2$ 1M ~ 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

Courtesy of Manning and Raghavan

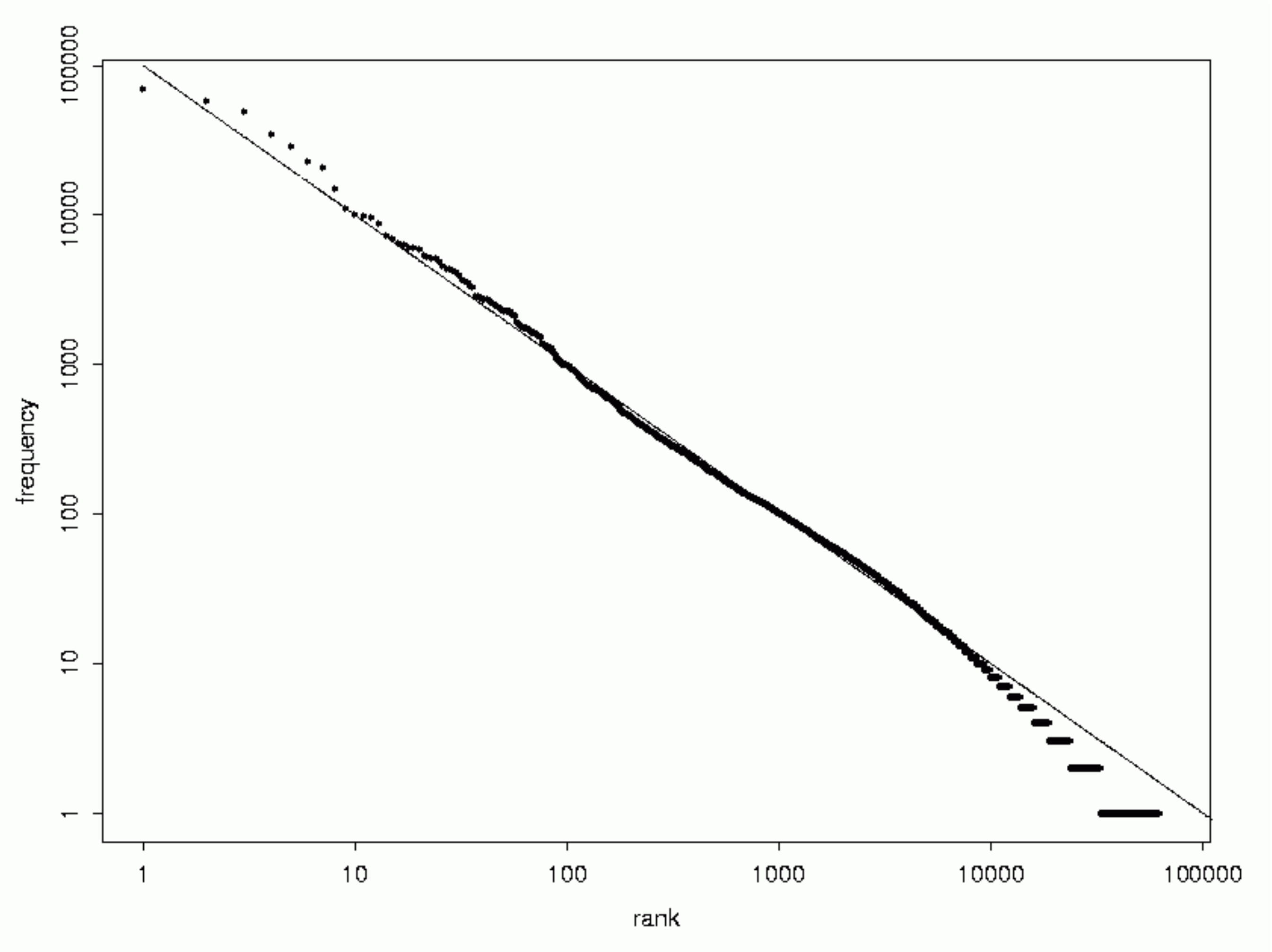
Zipf's law

- The *k*th most frequent term has frequency proportional to *1/k*.

- We use this for a crude analysis of the space used by our postings file pointers.
  - Not yet ready for analysis of dictionary space.

Courtesy of Manning and Raghavan

Zipfs law log-log plot



Courtesy of Manning and Raghavan

How big is the lexicon V?

- Grows (but more slowly) with corpus size
- Empirically okay model: Heap's Law

$$m = kT^b$$

Exercise: Can one derive this from Zipf's Law?

- where $b \approx 0.5$, $k \approx 30$–100; T = # tokens
- For instance TREC disks 1 and 2 (2 GB; 750,000 newswire articles): ≈ 500,000 terms
- *m* is decreased by case-folding, stemming
- Indexing all numbers could make it extremely large (so usually don't)
- Spelling errors contribute a fair bit of size

Courtesy of Manning and Raghavan

Module 3 Overview

- Review Inverted File Construction

- Chapter 5: Index Compression

  Zipf’s Law & Heap’s Law (IIR 5.1)

  Dictionaries (IIR 5.2)

  Inverted files (IIR 5.3)

Lexicons

- We want to store certain information for each word
  - The number of documents containing the word (aka document frequency)
  - An offset into a postings file
  - Often, we also want to know the total number of occurrences of the word
- We want minimal storage requirements and fast operations (insert, find)
  - While scanning documents
  - When processing user queries

How much memory do we need?

- Thought Experiment

  (following Managing Gigabytes, Chap. 4)

  - 1M word lexicon; 12 bytes for each value
  - 4-byte ints for DF, CF, & fileoffset

  12MB in ‘values’

  What about the keys?

- Approach #1: Fixed size strings

  Assume none is longer than 20 characters (or truncate)

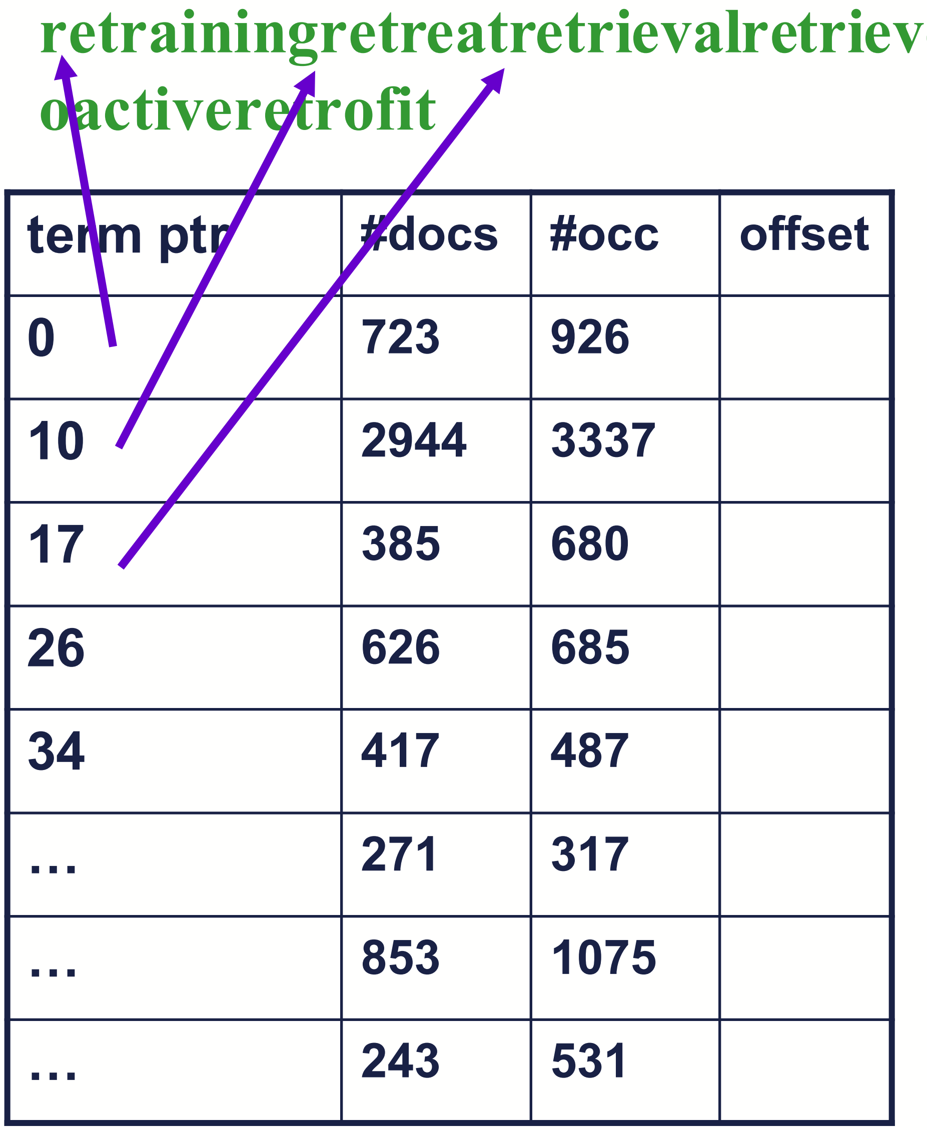
  *20 * 1M = 20 MB*

Fixed-size strings

	#docs	#occ	offset
retraining	723	926	
retreat	2944	3337	
retrieval	385	680	
retrieve	626	685	
retrieved	417	487	
retrieving	271	317	
retroactive	853	1075	
retrofit	243	531	

- TREC-8 Collection
  - 2 GB text
  - ~500k docs
  - ~500k unique terms
- Space for lexicon with 1M terms
  - 12 (vals) + 20 (keys) = 32 MB
- To find a query term
  - Load the array in memory
  - Use binary search
- **Can we do better?**

Terminated strings

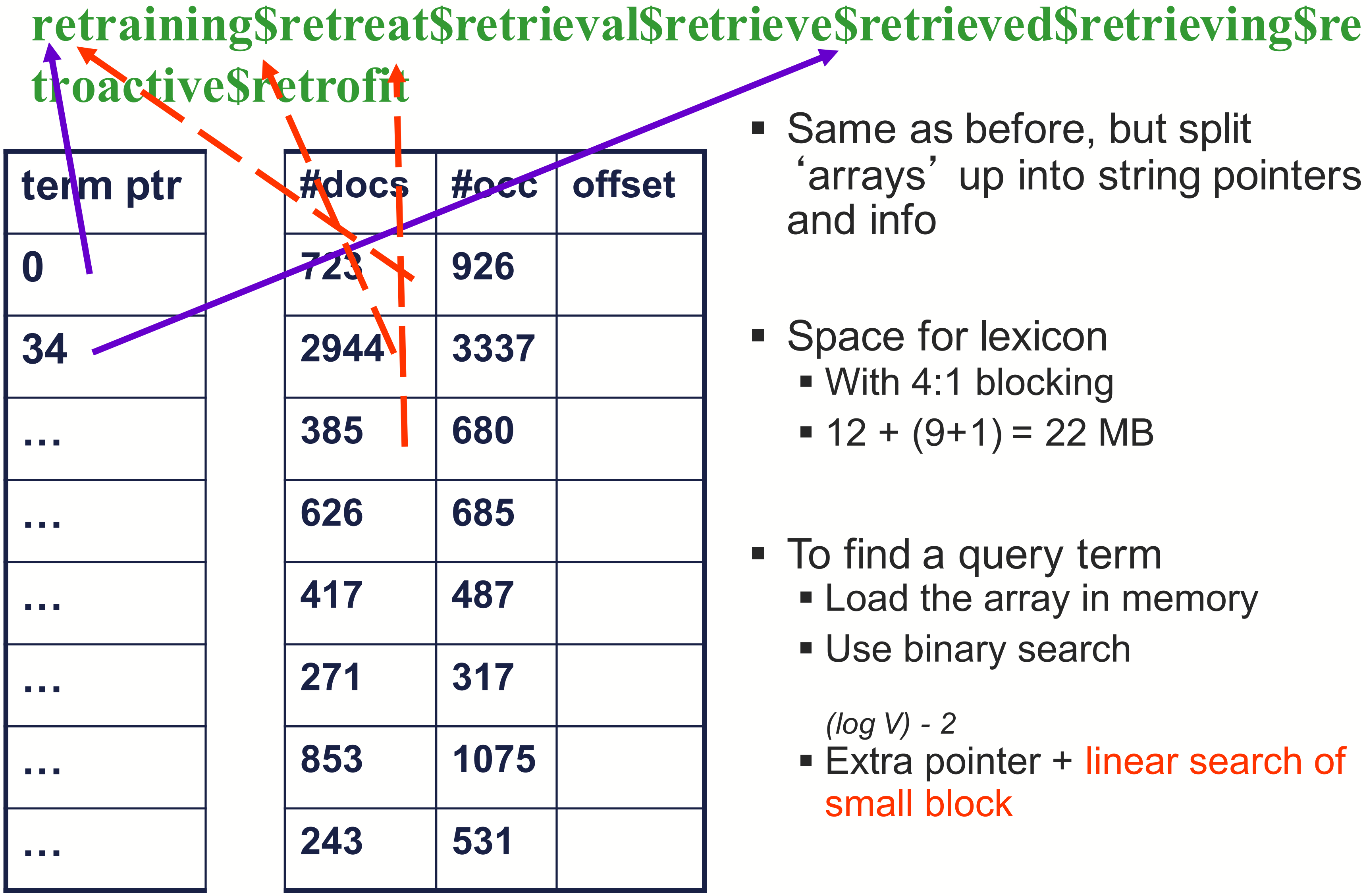


retrainingretreatretrievalretrieveretrievedretrievingretr
oactiveretrofit

term ptr	#docs	#occ	offset
0	723	926	
10	2944	3337	
17	385	680	
26	626	685	
34	417	487	
…	271	317	
…	853	1075	
…	243	531	

- Space for lexicon
  - Avg word len = ~8 chars
  - 12 + (9 + 4) = 25 MB (symbol)
  - 12 + (8+4) = 24 MB (subtract)
- To find a query term
  - Put string & array in memory
  - Use binary search
  - Follow extra pointer
- Can we do even better?

Terminated strings + Blocking



retraining$retreat$retrieval$retrieve$retrieved$retrieving$retroactive$retrofit

term ptr
0
34
…
…
…
…
…
…

#docs	#occ	offset
723	926	
2944	3337	
385	680	
626	685	
417	487	
271	317	
853	1075	
243	531	

shorter array (1/4)

- Same as before, but split 'arrays' up into string pointers and info

- Space for lexicon
  - With 4:1 blocking
  - 12 + (9+1) = 22 MB

- To find a query term
  - Load the array in memory
  - Use binary search

    (log V) - 2
  - Extra pointer + linear search of small block

- Can we do better still?

Term’ d strings; blocking; front-coding

retrainingretreatretrievalretrieveretrievedretrievingretroactiveretrofit

(retra) 5,5,ining 4,3,eat 4,5,ieval 7,1,e 8,1,d …, 0,1,s 1,2,ad

3-in-4 front-coding so binary search can be used

term ptr
0
34
…
…
…
…
…
…

#docs	#occ	offset
723	926	
2944	3337	
385	680	
626	685	
417	487	
271	317	
853	1075	
243	531	

- Same as before, but realize lots of repetition in ‘key space’

- Space savings
  - For English lexicons, 40%
  - 12 + (5+1) = 18 MB

- To find a query term
  - Load the array in memory
  - Use binary search
  - Extra pointer + linear search of small block + tiny extra effort to decode

- Can we do better?

Minimal-Perfect Hash Functions

- Hashtables maps k keys onto integers 0 through m
  - For m>=k, as m approaches k, collisions increase
  - Birthday paradox
- Perfect hash function maps k keys onto k integers without collisions
  - *Minimal* perfect hashing maps k keys onto integers 0 to k-1
- Can such a function be found
  - Yes, if set of keys is fixed, using fast, probabilistic algorithm

    In fact, can even find functions that preserve the order of the input keys! (a = 0, … zymurgy= k-1)

    What is the catch?

The hash *function* takes up space

- For an order preserving function
- At least, n log(n) bits are required for n keys
- For 1 M keys, that’s 1.25 bytes / key
  - 5 MB in space vs. previous 6 MB
  - Starting to spend a lot of effort for small gains

Really Large Lexicons

- Chinese is a non-alphabetic language
  - O(10k) symbols
- N-gram indexing represents ‘terms’ as 3-character sequences
  - 10,000^3 = 10^12 terms (1,000 billion)
- Instead of in-memory, go to external storage for lexicon
  - Slows down indexing significantly
  - But, caching can help
  - Same idea as databases, use B+-trees

Module 3 Overview

- Review Inverted File Construction

- Chapter 5: Index Compression

  Zipf's Law & Heap's Law (IIR 5.1)

  Dictionaries (IIR 5.2)

  Inverted files (IIR 5.3)

Index Compression

- In general, an index requires less space than a text
  - Even for a comprehensive index

    Rule of thumb: 10-30%
  - Thought question: why does it take up less space?

- Why reduce the size of an inverted file?
  - We can't fit it all in memory like a lexicon
  - We are trading more time for reduced space and we are happy – what is gained?

This is a good question to think on. The next slide has a answer…

Why less is more

- Save on extra disk space (electricity, maintenance)
  - Less space does save dollars. Fewer devices, less electricity, less maintenance and labor.
- Smaller postings, less disk transfer time
  - If we create smaller files we spend less time doing IO operations.
- Faster indexing
  - For Algorithm E, we can fit more data in memory if it is compressed; this lets us write out fewer files and do fewer n-way merges
- Caching during query processing
  - We can't fit all of an index in memory. But if the index is compressed, we can fit some terms in memory, and this can speed up some search queries.

Compressing Integers

- Binary Warm-Up
- Binary representation
  - Better than ASCII!
- Three variable-bit coding schemes
  - Unary code
  - Gamma code
  - Delta code

Encoding Numbers in Binary

- Representing numbers as character symbols is inefficient

  '7856' in ASCII is: 0110111 0111000 0110101 0110110_2 (28 bits)

  7856_{10} in binary is: 1111010110000_2 (13 bits)

- Example: small binary numbers

	$2^4 = 16$	$2^3=8$	$2^2 = 4$	$2^1 = 2$	$2^0 = 1$
0	0	0	0	0	0
1	0	0	0	0	1
2	0	0	0	1	0
3	0	0	0	1	1
4	0	0	1	0	0
5	0	0	1	0	1
6	0	0	1	1	0
7	0	0	1	1	1
8	0	1	0	0	0
9	0	1	0	0	1
16	1	0	0	0	0
31	1	1	1	1	1

$$3 = (0 \text{ x } 2^4) + (0 \text{ x } 2^3) + (0 \text{ x } 2^2) + (1 \text{ x } 2^1) + (1 \text{ x } 2^0)$$
$$= 0 + 0 + 0 + 2 + 1$$

$$9 = (0 \text{ x } 2^4) + (1 \text{ x } 2^3) + (0 \text{ x } 2^2) + (0 \text{ x } 2^1) + (1 \text{ x } 2^0)$$
$$= 0 + 8 + 0 + 0 + 1$$

What are we compressing?

- Say a term occurs in a list of documents:
  - 3, 17, 32, 100, 105, 1000, 2000
- We can map these numbers to smaller values: gaps

  3,14,15,68,5,895,1000
  - No loss of information!

  Now, consider a frequent term, it occurs in almost every other document

  …, 1, 2, 3, 2, 1, 1, 2, 3, 1, 2, 7, 2, 1, …
  - These smaller numbers are much small, therefore, more compressible

  Only a few can be really big

Unary Coding

- Only for integers greater than or equal to zero (MG: one)

- Represent x as:

  x one bits, followed by a single zero bit

STOP!!! DON'T DO IT! Wikipedia and other books explain the following coding schemes using different notation and different representations. I'm teaching this the textbook's way, and you need to know this way for the homework and exams. You've been warned!

  1000 '1's followed by a 0

  - unary(1 million) = don't ask!

- Code is only 'good' for small numbers!

Elias's Gamma Code (1975)

- Represent x in two parts:

  unary(floor(log(x))) followed by

  x-$2^{(floor(log\ x))}$ in binary

  floor(log(x)) is the number of binary 'suffix' bits; <u>use base 2</u>

- gamma(2) = 10 followed by 0 = 10 0
- gamma(5) = unary(2) followed by binary(5-2^2)
  = unary(2) and binary(1)
  = 110 01
- gamma(100) = unary(6) and binary(36) in 6 bits

  Only takes 13 bits

  gamma(1 million) only requires 39 bits

Decoding a Gamma Code

- Input: 1 1 1 1 0 0 1 0 1
- We first read a unary code. So count the number of one bits until we see a zero.
  - 4 one bits
- So the answer is 2^4 + “0 1 0 1” in binary
  - 16 + 5 = 21

Elias’s Delta Code (1975)

- (See exercise IIR 5.9)
- Almost exactly like previous method
  - Except, store the prefix (i.e., the number of binary suffix bits) using the Gamma code
- Represent x in two parts:

gamma(floor(log(x))) followed by

$x-2^{(floor(log\ x))}$ in binary – using floor(log x) bits

- delta(2) = gamma(1)=0 followed by 0 = 0 0
- delta(5) = gamma(2) followed by binary($5-2^2$)
  = gamma(2) and binary(1)
  = 100 01
- delta(100) = gamma(6) and binary(36) in 6 bits

Only takes 11 bits

delta(1 million) only requires 28 bits

What difference does it make?

Method	Bible (KJV) (N=~30,000)	TREC Disks 1-3 (N=~720,000)
Unary	262	1918
Binary	15	20
Gamma	6.51	6.63
Delta	6.23	6.38
Local Interpolative	5.24	5.18

Bits per posting. Based on Managing Gigabytes 2nd, pg 129

What about the term frequency?

- We just explained how to encode document ids
  - Using gaps

- But postings lists are usually pairs
  - (docid, freq), (docid, freq), (docid, freq), ...

- How should we encode term frequencies with each document?

Byte-aligned compression

- Used by many commercial/research systems
  - Good low-tech blend of variable-length coding and sensitivity to alignment issues

- Fix a word-width of, here, $w = 8$ bits.
- Dedicate 1 bit (high bit) to be a *continuation bit c.*
- If the gap G fits within $(w - 1) = 7$ bits, binary-encode it in the 7 available bits and set $c = 0$.
- Else set $c = 1$, encode low-order $(w - 1)$ bits, and then use one or more additional words to encode $\lfloor G/2^{w-1} \rfloor$ using the same algorithm

Courtesy of Manning and Raghavan

Word-aligned binary codes

- More complex schemes – indeed, ones that respect 32-bit word alignment – are possible
  - Byte alignment is especially inefficient for very small gaps (such as for commonest words)
- Say we now use 32 bit word with 2 control bits
- Sketch of an approach:
  - If the next 30 gaps are 1 or 2 encode them in binary within a single word
  - If next gap > 2^{15}, encode just it in a word
  - For intermediate gaps, use intermediate strategies
  - Use 2 control bits to encode coding strategy

Courtesy of Manning and Raghavan

Docid Reassignment

- Normally docids are assigned in the same order that documents are scanned for indexing
  - For indexing, the ordering is arbitrary
- Some assignments of ids to documents are better than others
  - Average gap length can be reduced!

D1: Dewey defeats Truman
D2: Berlin airlift prevents starvation
D3: Truman defeats Dewey
D4: Eisenhower leaves Berlin to Red Army

Berlin	2	4
Dewey	1	3
Truman	1	3

D1: Dewey defeats Truman
D2: Truman defeats Dewey
D3: Berlin airlift prevents starvation
D4: Eisenhower leaves Berlin to Red Army

Berlin	3	4
Dewey	1	2
Truman	1	2

d-gap Compression

- TSP approximation
  - Shieh et al., ‘Inverted file compression through document identifier reassignment’, Information Processing and Management, 39(1), 117-131, 2003
  - 15-20% reduction seems possible

- Greedy algorithm
  - Chris Buckley on TREC Terabyte corpus

Dual Files

- Inverted files are term-referenced lists of docids and term-counts
  - A docid-referenced list of termids and term-counts is a 'dual file'
  - Like a column of the term-document matrix instead of a row
  - The same compression techniques can be applied

- Uses
  - for finding all terms that appear in a given document
  - to compare two documents directly

Dual Files cont’d

- Access to each ‘vector’ requires a different disk seek
  - Thus can not afford to do many

- Application
  - Finding expansion terms for a query (aka, blind relevance feedback)
  - Take the top 10 ranked documents and look for terms that might be useful to augment the original query with

  Then do a second pass of retrieval