BLG 540E
TEXT RETRIEVAL SYSTEMS

Index Construction and Compression

Arzucan Özgür

Faculty of Computer and Informatics, İstanbul Techical University

March 4, 2011

Index Construction

Index construction

- How do we construct an index?
- What strategies can we use with limited main memory?

Hardware basics

- Many design decisions in information retrieval are based on the characteristics of hardware
- We begin by reviewing hardware basics

Hardware basics

- Access to data in memory is much faster than access to data on disk.
- Disk seeks: No data is transferred from disk while the disk head is being positioned.
- Therefore: Transferring one large chunk of data from disk to memory is faster than transferring many small chunks.
- Disk I/O is block-based: Reading and writing of entire blocks (as opposed to smaller chunks).
- Block sizes: 8KB to 256 KB.

Hardware basics

- Servers used in IR systems now typically have several GB of main memory, sometimes tens of GB.
- Available disk space is several orders of magnitude larger.
- Fault tolerance is very expensive: It's much cheaper to use many regular machines rather than one fault tolerant machine.

RCV1: Our collection for this lecture

- Shakespeare's collected works definitely aren't large enough for demonstrating many of the points in this course.
- The collection we'll use isn't really large enough either, but it's publicly available and is at least a more plausible example.
- As an example for applying scalable index construction algorithms, we will use the Reuters RCV1 collection.
- This is one year of Reuters newswire (part of 1995 and 1996)

A Reuters RCV1 document

REUTERS

You are here: Home > News > Science > Article

Go to a Section: U.S. International Business Markets Politics Entertainment Technology Sports Oddly Enoug

Extreme conditions create rare Antarctic clouds

Tue Aug 1, 2006 3:20am ET

Email This Article | Print This Article | Reprints

[-] Text [+]

SYDNEY (Reuters) - Rare, mother-of-pearl colored clouds caused by extreme weather conditions above Antarctica are a possible indication of global warming, Australian scientists said on Tuesday.

Known as nacreous clouds, the spectacular formations showing delicate wisps of colors were photographed in the sky over an Australian meteorological base at Mawson Station on July 25.

Reuters RCV1 statistics

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
	non-positional postings	100,000,000

Recall sort-based index construction (Lec1)

- Documents are parsed to extract words and these are saved with the Document ID.

Doc 1

I did enact Julius Caesar I was killed i' the Capitol; Brutus killed me.

Doc 2

So let it be with Caesar. The noble Brutus hath told you Caesar was ambitious

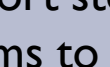
Term	Doc #
I	1
did	1
enact	1
julius	1
caesar	1
I	1
was	1
killed	1
i'	1
the	1
capitol	1
brutus	1
killed	1
me	1
so	2
let	2
it	2
be	2
with	2
caesar	2
the	2
noble	2
brutus	2
hath	2
told	2
you	2
caesar	2
was	2
ambitious	2

Key step

- After all documents have been parsed, the inverted file is sorted by terms.

We focus on this sort step.
We have 100M items to sort.

Term	Doc #
I	1
did	1
enact	1
julius	1
caesar	1
I	1
was	1
killed	1
i'	1
the	1
capitol	1
brutus	1
killed	1
me	1
so	2
let	2
it	2
be	2
with	2
caesar	2
the	2
noble	2
brutus	2
hath	2
told	2
you	2
caesar	2
was	2
ambitious	2



Term	Doc #
ambitious	2
be	2
brutus	1
brutus	2
capitol	1
caesar	1
caesar	2
caesar	2
did	1
enact	1
hath	1
I	1
I	1
i'	1
it	2
julius	1
killed	1
killed	1
let	2
me	1
noble	2
so	2
the	1
the	2
told	2
you	2
was	1
was	2
with	2

Scaling index construction

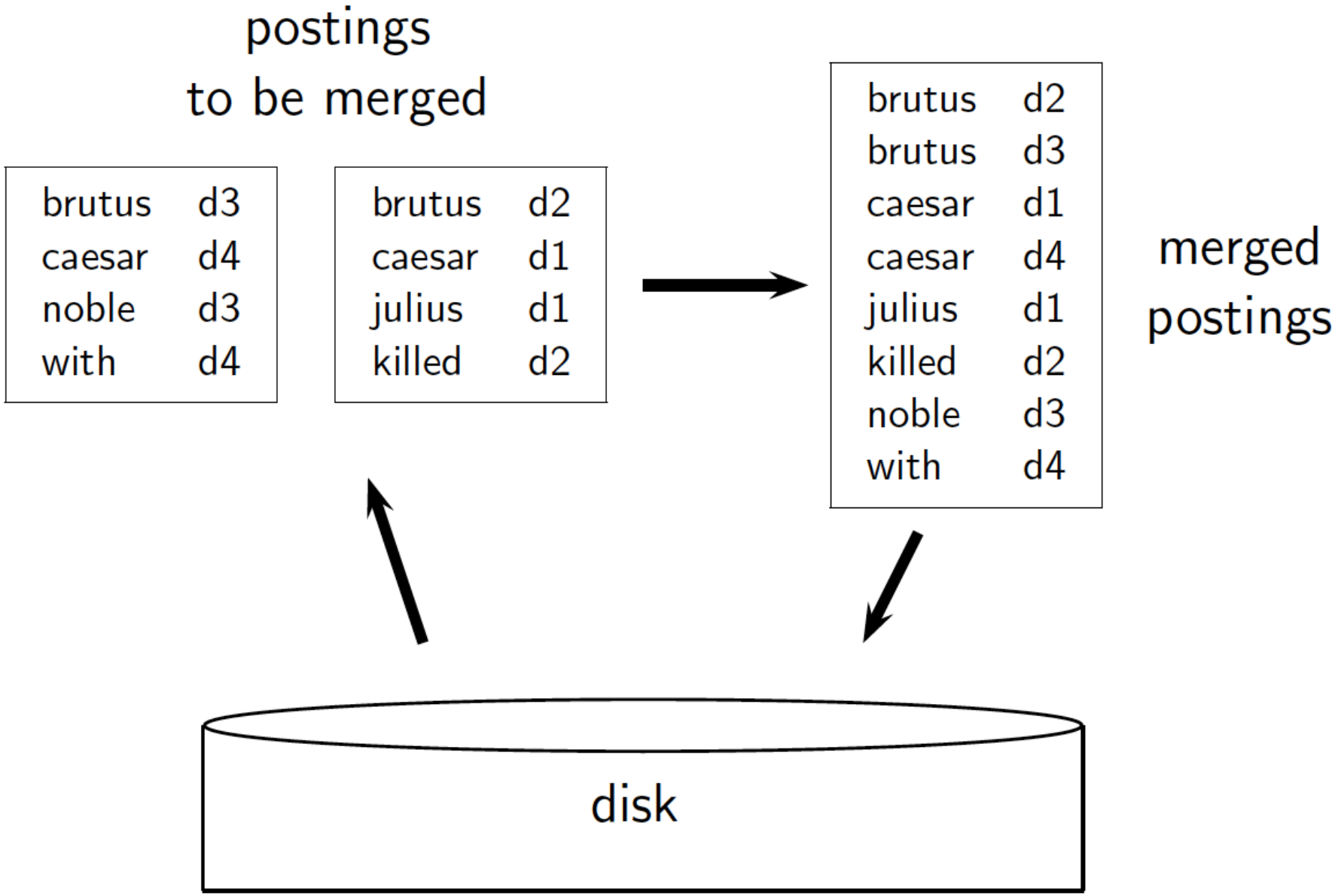
- In-memory index construction does not scale.
- How can we construct an index for very large collections?
- Taking into account the hardware constraints we just learned about . . .
- Memory, disk, speed, etc.

Sort-based index construction

- As we build the index, we parse docs one at a time.
  - While building the index, we cannot easily exploit compression tricks (you can, but much more complex)
- The final postings for any term are incomplete until the end.
- At 12 bytes per non-positional postings entry *(term, doc, freq)*, demands a lot of space for large collections.
- T = 100,000,000 in the case of RCV1
  - So … we can do this in memory, but typical collections are much larger. E.g. the *New York Times* provides an index of >150 years of newswire
- Thus: We need to store intermediate results on disk. (Need to use an external sorting algorithm).

BSBI: Blocked sort-based Indexing (Sorting with fewer disk seeks)

- 12-byte (4+4+4) records *(term, doc, freq)*.
- These are generated as we parse docs.
- Must now sort 100M such 12-byte records by *term*.
- Define a Block ~ 10M such records
  - Can easily fit a couple into memory.
  - Will have 10 such blocks to start with.
- Basic idea of algorithm:
  - Accumulate postings for each block, sort, write to disk.
  - Then merge the blocks into one long sorted order.


postings
to be merged
brutus d3
caesar d4
noble d3
with d4
brutus d2
caesar d1
julius d1
killed d2
brutus d2
brutus d3
caesar d1
caesar d4
julius d1
killed d2
noble d3
with d4
merged
postings
disk

Blocked sort-based Indexing

```
BSBIndexConstruction()
 1  n ← 0
 2  while (all documents have not been processed)
 3  do n ← n + 1
 4       block ← ParseNextBlock()
 5       BSBI-Invert(block)
 6       WriteBlockToDisk(block, f_n)
 7  MergeBlocks(f_1, ..., f_n; f_merged)
```

Problem with sort-based algorithm

- Our assumption was: we can keep the dictionary in memory.
- We need the dictionary (which grows dynamically) in order to implement a term to termID mapping.
- Actually, we could work with term,docID postings instead of termID,docID postings . . .
- . . . but then intermediate files become very large. (We would end up with a scalable, but very slow index construction method.)

SPIMI: Single-pass in-memory indexing

- Key idea 1: Generate separate dictionaries for each block – no need to maintain term-termID mapping across blocks.
- Key idea 2: Don't sort. Accumulate postings in postings lists as they occur.
- With these two ideas we can generate a complete inverted index for each block.
- These separate indexes can then be merged into one big index.

SPIMI-Invert

```
SPIMI-INVERT(token_stream)
 1  output_file = NEWFILE()
 2  dictionary = NEWHASH()
 3  while (free memory available)
 4  do token ← next(token_stream)
 5      if term(token) ∉ dictionary
 6          then postings_list = ADDTODICTIONARY(dictionary, term(token))
 7          else postings_list = GETPOSTINGSLIST(dictionary, term(token))
 8      if full(postings_list)
 9          then postings_list = DOUBLEPOSTINGSLIST(dictionary, term(token))
10      ADDTOPOSTINGSLIST(postings_list, docID(token))
11  sorted_terms ← SORTTERMS(dictionary)
12  WRITEBLOCKTODISK(sorted_terms, dictionary, output_file)
13  return output_file
```

- Merging of blocks is analogous to BSBI.

SPIMI: Compression

- Compression makes SPIMI even more efficient.
  - Compression of terms
  - Compression of postings

Distributed indexing

- For web-scale indexing:

  must use a distributed computing cluster
- Individual machines are fault-prone
  - Can unpredictably slow down or fail
- How do we exploit such a pool of machines?

Google data centers

- Google data centers mainly contain commodity machines.
- Data centers are distributed around the world.
- Estimate: a total of 1 million servers, 3 million processors/cores (Gartner 2007)
- Estimate: Google installs 100,000 servers each quarter.
  - Based on expenditures of 200–250 million dollars per year
- This would be 10% of the computing capacity of the world!?!

Distributed indexing

- Maintain a *master* machine directing the indexing job – considered "safe".
- Break up indexing into sets of (parallel) tasks.
- Master machine assigns each task to an idle machine from a pool.

Parallel tasks

- We will use two sets of parallel tasks
  - Parsers
  - Inverters
- Break the input document collection into *splits*
- Each split is a subset of documents (corresponding to blocks in BSBI/SPIMI)

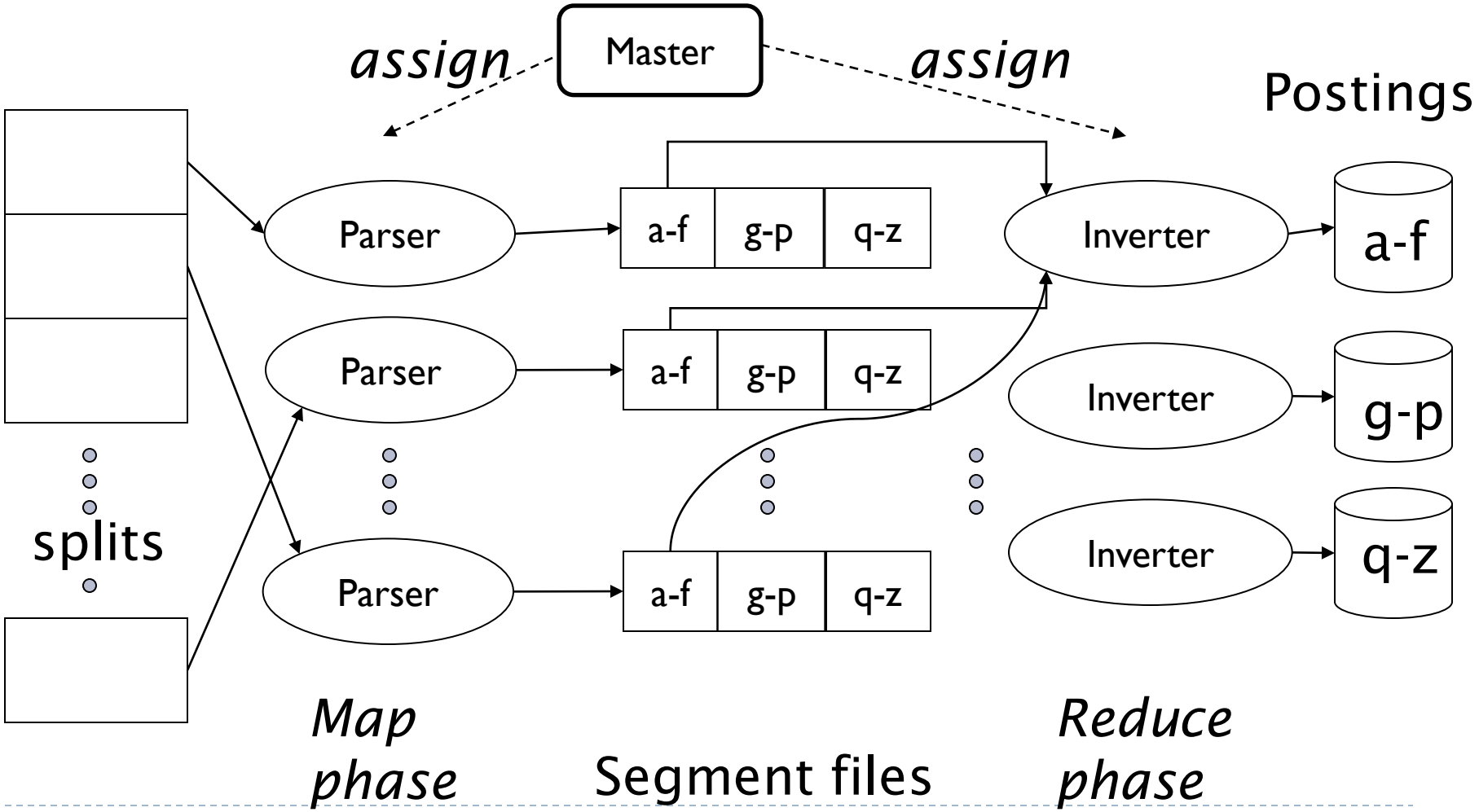
Parsers

- Master assigns a split to an idle parser machine
- Parser reads a document at a time and emits (term, doc) pairs
- Parser writes pairs into *j* partitions
- Each partition is for a range of terms' first letters
  - (e.g., ***a-f, g-p, q-z***) – here *j* = 3.
- Now to complete the index inversion

Inverters

- An inverter collects all (term,doc) pairs (= postings) for one term-partition.
- Sorts and writes to postings lists

Data flow


assign
Master
assign
Postings
Parser
Parser
Parser
a-f
g-p
q-z
a-f
g-p
q-z
a-f
g-p
q-z
Inverter
Inverter
Inverter
a-f
g-p
q-z
splits
Map phase
Segment files
Reduce phase

MapReduce

- The index construction algorithm we just described is an instance of MapReduce.
- MapReduce (Dean and Ghemawat 2004) is a robust and conceptually simple framework for distributed computing.
- The Google indexing system (ca. 2002) as consisted of a number of phases, each implemented in MapReduce.
  - Index construction was just one phase.
  - Another phase: transforming a term-partitioned index into a document-partitioned index.
    - *Term-partitioned:* one machine handles a subrange of terms
    - *Document-partitioned:* one machine handles a subrange of documents
  - Most search engines use a document-partitioned index … better load balancing, etc.

Index construction in MapReduce

Schema of map and reduce functions

map:	input	$\rightarrow$ list(k, v)
reduce:	(k,list(v))	$\rightarrow$ output

Instantiation of the schema for index construction

map:	web collection	$\rightarrow$ list(termID, docID)
reduce:	($\langle$termID$_1$, list(docID)$\rangle$, $\langle$termID$_2$, list(docID)$\rangle$, ...)	$\rightarrow$ (postings_list$_1$, postings_list$_2$, ...)

Example for index construction

map:	d_2 : C DIED. d_1 : C CAME, C C'ED.	$\rightarrow$ ($\langle$C, $d_2\rangle$, $\langle$DIED,$d_2\rangle$, $\langle$C,$d_1\rangle$, $\langle$CAME,$d_1\rangle$, $\langle$C,$d_1\rangle$, $\langle$C'ED,$d_1\rangle$)
reduce:	($\langle$C,(d_2,d_1,d_1)$\rangle$,$\langle$DIED,(d_2)$\rangle$,$\langle$CAME,(d_1)$\rangle$,$\langle$C'ED,(d_1)$\rangle$)	$\rightarrow$ ($\langle$C,(d_1:2,d_2:1)$\rangle$,$\langle$DIED,(d_2:1)$\rangle$,$\langle$CAME,(d_1:1)$\rangle$,$\langle$C'ED,(d_1:1)$\rangle$)

Dynamic indexing

- Up to now, we have assumed that collections are static.
- They rarely are:
  - Documents come in over time and need to be inserted.
  - Documents are deleted and modified.
- This means that the dictionary and postings lists have to be modified:
  - Postings updates for terms already in dictionary
  - New terms added to dictionary

Simplest approach

- Maintain "big" main index
- New docs go into "small" auxiliary index
- Search across both, merge results
- Deletions
  - Invalidation bit-vector for deleted docs
  - Filter docs output on a search result by this invalidation bit-vector
- Periodically, re-index into one main index

Issues with main and auxiliary indexes

- Problem of frequent merges
- Poor performance during merge
- Actually:
  - Merging of the auxiliary index into the main index is efficient if we keep a separate file for each postings list.
  - Merge is the same as a simple append.
  - But then we would need a lot of files – inefficient for O/S.
- Assumption for the rest of the lecture: The index is one big file.
- In reality: Use a scheme somewhere in between (e.g., split very large postings lists, collect postings lists of length 1 in one file etc.)

Further issues with multiple indexes

- Collection-wide statistics are hard to maintain
- E.g., when we spoke of spell-correction: which of several corrected alternatives do we present to the user?
  - We said, pick the one with the most hits
- How do we maintain the top ones with multiple indexes and invalidation bit vectors?
  - One possibility: ignore everything but the main index for such ordering

Dynamic indexing at search engines

- All the large search engines now do dynamic indexing
- Their indices have frequent incremental changes
  - News items, blogs, new topical web pages
    - Sarah Palin, …
- But (sometimes/typically) they also periodically reconstruct the index from scratch
  - Query processing is then switched to the new index, and the old index is then deleted

Building Positional Indexes

- Basically the same problem except that the intermediate data structures are large.

Index 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

Effect of preprocessing

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

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.

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.
- In practice, the vocabulary will keep growing with the collection size

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 ½
  - It is the simplest possible relationship between the two in log-log space
  - An empirical finding ("empirical law")

Heaps' Law

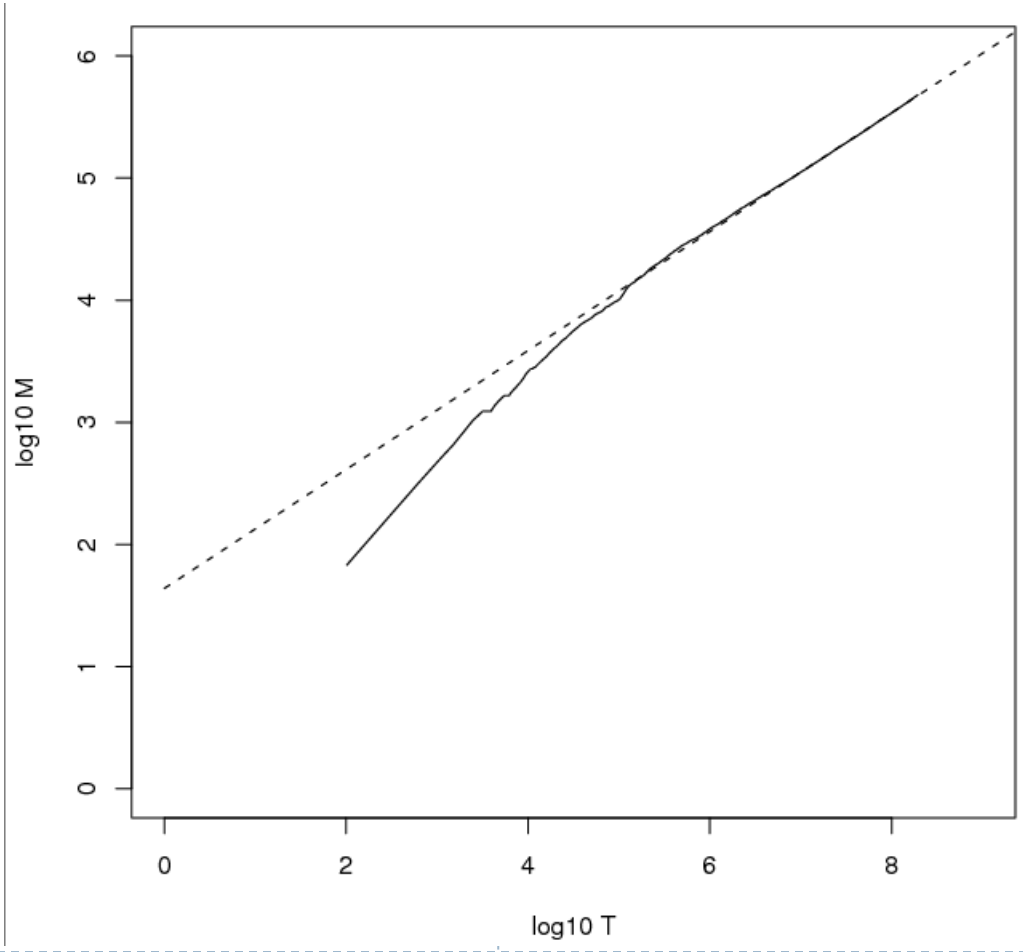
For RCV1, the dashed line

Good empirical fit for Reuters RCV1 !

For first 1,000,020 tokens,

law predicts 38,323 terms;

actually, 38,365 terms



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.

Word distributions

- Words are not distributed evenly!
- Same goes for letters of the alphabet (ETAOIN SHRDLU), city sizes, wealth, etc.
- Usually, the 80/20 rule applies (80% of the wealth goes to 20% of the people or it takes 80% of the effort to build the easier 20% of the system)…

Shakespeare

- ## Romeo and Juliet:

- And, 667; The, 661; I, 570; To, 515; A, 447; Of, 382; My, 356; Is, 343; That, 343; In, 314; You, 289; Thou, 277; Me, 262; Not, 257; With, 234; It, 224; For, 223; This, 215; Be, 207; But, 181; Thy, 167; What, 163; O, 160; As, 156; Her, 150; Will, 147; So, 145; Thee, 139; Love, 135; His, 128; Have, 127; He, 120; Romeo, 115; By, 114; She, 114; Shall, 107; Your, 103; No, 102; Come, 96; Him, 96; All, 92; Do, 89; From, 86; Then, 83; Good, 82; Now, 82; Here, 80; If, 80; An, 78; Go, 76; On, 76; I'll, 71; Death, 69; Night, 68; Are, 67; More, 67; We, 66; At, 65; Man, 65; Or, 65; There, 64; Hath, 63; Which, 60;
- …
- A-bed, 1; A-bleeding, 1; A-weary, 1; Abate, 1; Abbey, 1; Abhorred, 1; Abhors, 1; Aboard, 1; Abound'st, 1; Abroach, 1; Absolved, 1; Abuse, 1; Abused, 1; Abuses, 1; Accents, 1; Access, 1; Accident, 1; Accidents, 1; According, 1; Accursed, 1; Accustom'd, 1; Ache, 1; Aches, 1; Aching, 1; Acknowledge, 1; Acquaint, 1; Acquaintance, 1; Acted, 1; Acting, 1; Action, 1; Acts, 1; Adam, 1; Add, 1; Added, 1; Adding, 1; Addle, 1; Adjacent, 1; Admired, 1; Ado, 1; Advance, 1; Adversary, 1; Adversity's, 1; Advise, 1; Afeard, 1; Affecting, 1; Afflicted, 1; Affliction, 1; Affords, 1; Affray, 1; Affright, 1; Afire, 1; Agate-stone, 1; Agile, 1; Agree, 1; Agrees, 1; Aim'd, 1; Alderman, 1; All-cheering, 1; All-seeing, 1; Alla, 1; Alliance, 1; Alligator, 1; Allow, 1; Ally, 1; Although, 1;

http://www.mta75.org/curriculum/english/Shakes/index.html
(visited in Dec. 2006)

The BNC (Adam Kilgarriff)

- 1 6187267 the det
- 2 4239632 be v
- 3 3093444 of prep
- 4 2687863 and conj
- 5 2186369 a det
- 6 1924315 in prep
- 7 1620850 to infinitive-marker
- 8 1375636 have v
- 9 1090186 it pron
- 10 1039323 to prep
- 11 887877 for prep
- 12 884599 i pron
- 13 760399 that conj
- 14 695498 you pron
- 15 681255 he pron
- 16 680739 on prep
- 17 675027 with prep
- 18 559596 do v
- 19 534162 at prep
- 20 517171 by prep

Kilgarriff, A. Putting Frequencies in the Dictionary.
International Journal of Lexicography
10 (2) 1997. Pp 135--155

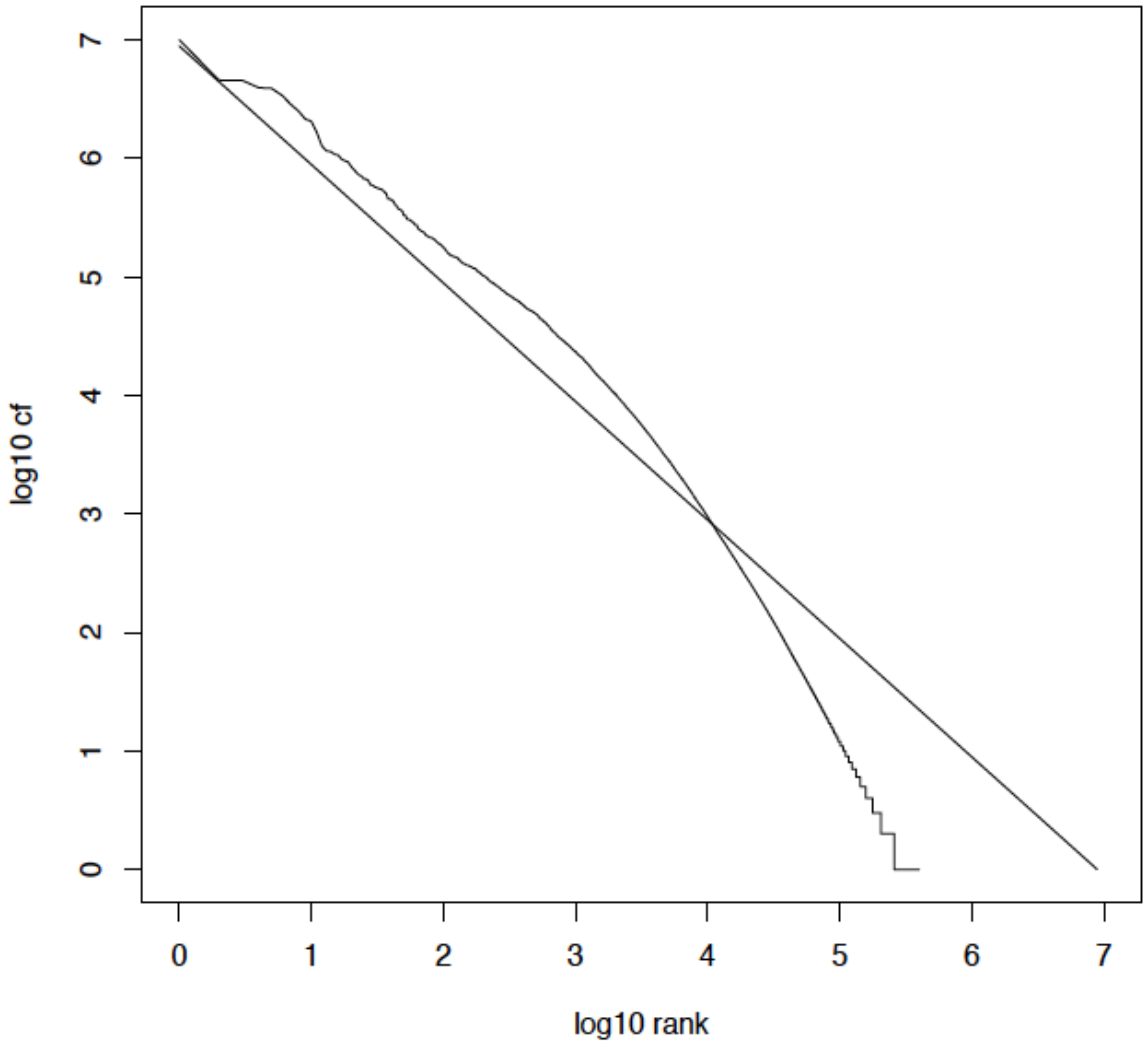
Stop words

- 250-300 most common words in English account for 50% or more of a given text.
- Example: “the” and “of” represent 10% of tokens. “and”, “to”, “a”, and “in” - another 10%. Next 12 words - another 10%.
- Moby Dick Ch.1: 859 unique words (types), 2256 word occurrences (tokens). Top 65 types cover 1132 tokens (> 50%).

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

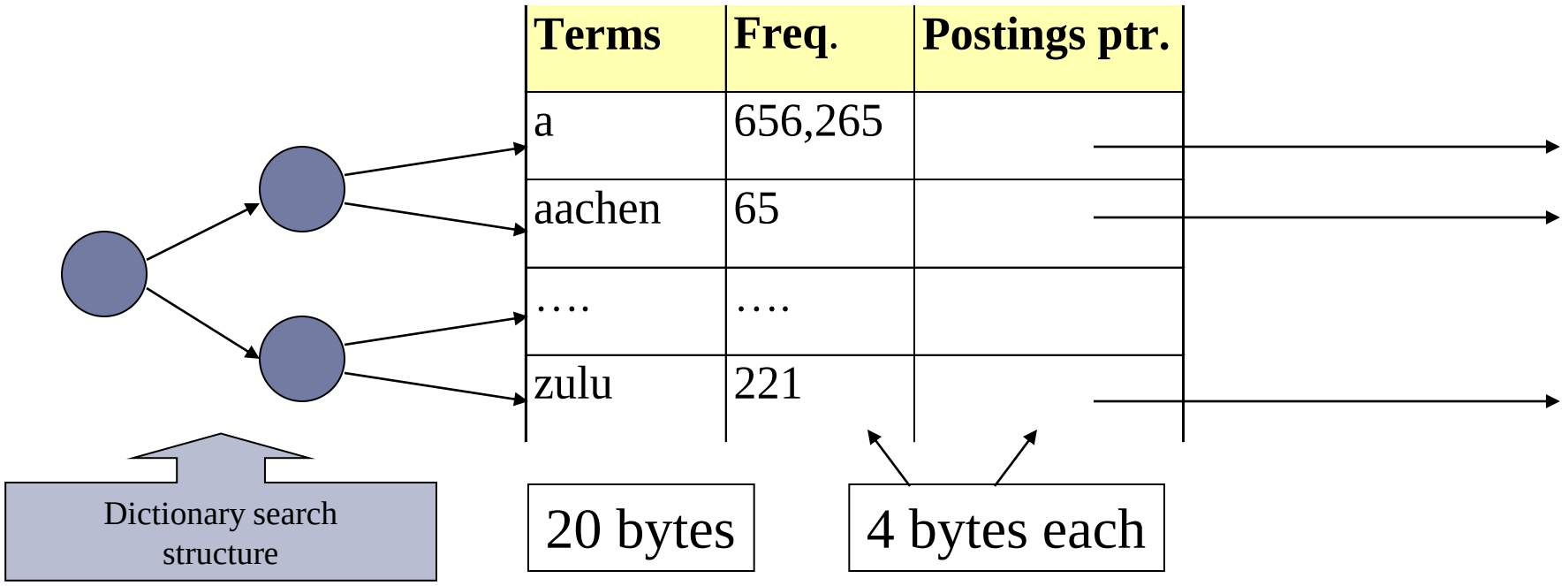
DICTIONARY COMPRESSION

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.

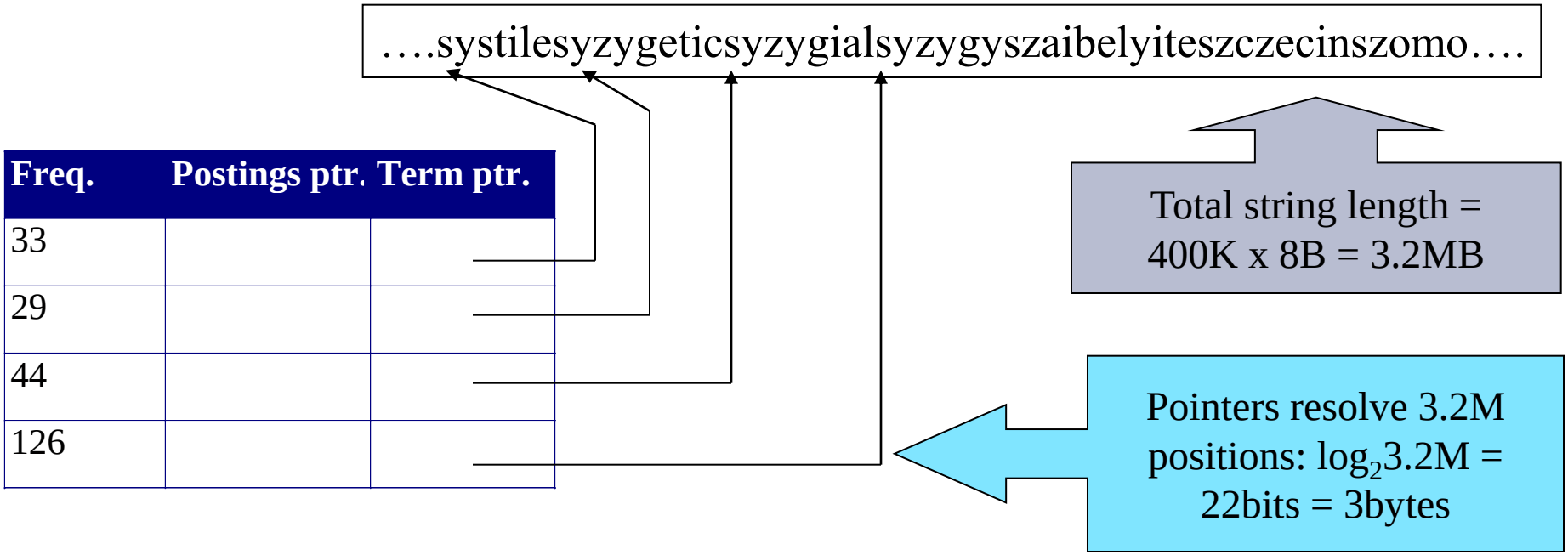


Fixed-width terms are wasteful

- 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*.
- Ave. dictionary word in English: ~8 characters

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.

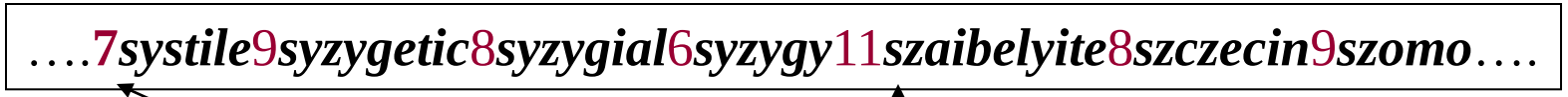


Space for dictionary as a string

- 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 $\Rightarrow$ 7.6 MB (against 11.2MB for fixed width)

Blocking

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



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

Save 9 bytes on 3 pointers.

Lose 4 bytes on term lengths.

Net

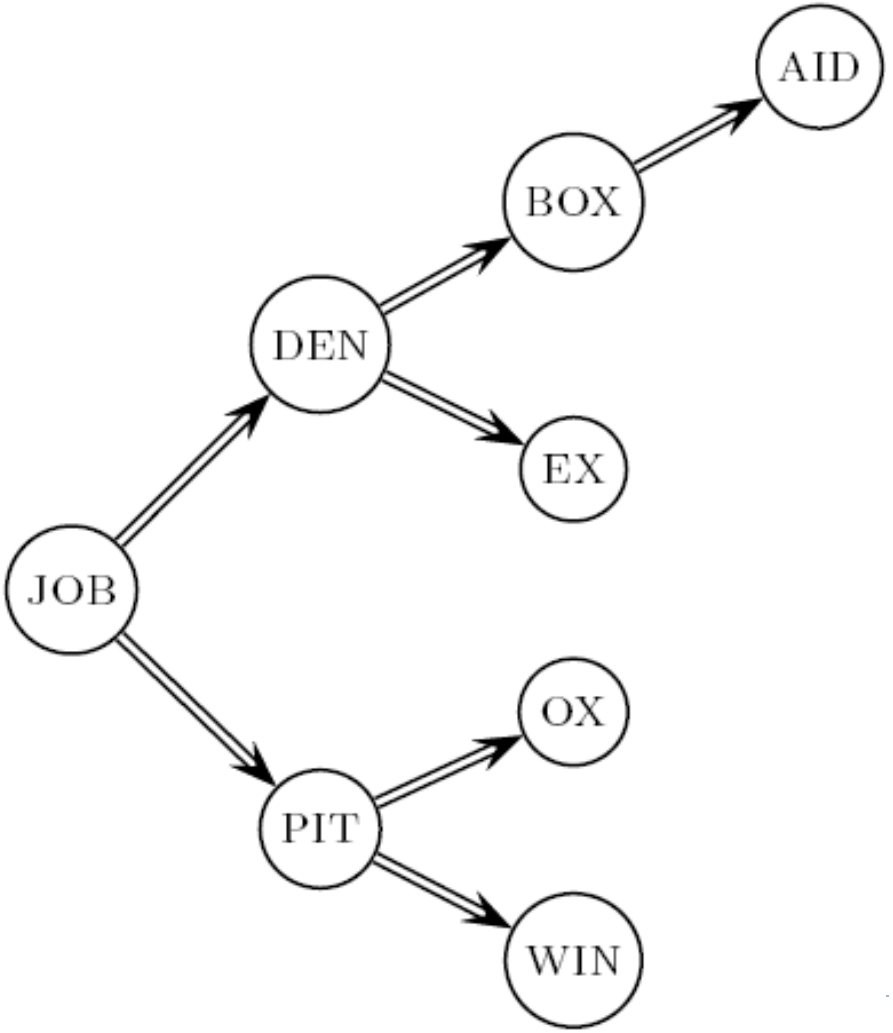
- Example for block size $k = 4$
- Save 5 bytes per four-term block.
- Total: 400,000/4 * 5 = 0.5 MB

Saved another ~0.5MB. This reduces the size of the dictionary from 7.6 MB to 7.1 MB.
We can save more with larger *k*.

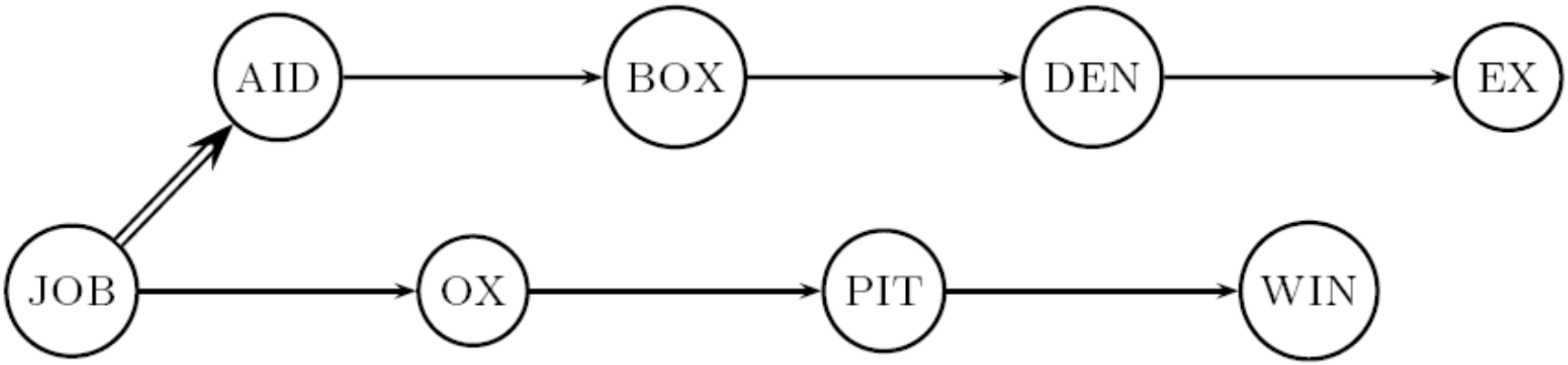
Why not go with larger *k*?

Dictionary search without blocking

- Assuming each dictionary term 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



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

Front coding

- Front-coding:
  - Sorted words commonly have long common prefix – store differences only

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

→8*automat***a*1◊*e*2◊*ic*3◊*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

Fixed length codes

- Binary representations
  - ASCII
  - Representational power (2^k symbols where k is the number of bits)

Variable length codes

- Alphabet:

A	.-	N	-.	0	-----		
B	-...	O	---	1	.----		
C	-.-.	P	.--.	2	..---		
D	-..	Q	--.-	3	...—		
E	.	R	.-.	4	....-		
F	..-.	S	...	5	.....		
G	--.	T	-	6	-....		
H	....	U	..-	7	--...		
I	..	V	...-	8	---..		
J	.---	W	.--	9	----.		
K	-.-	X	-..-				
L	.-..	Y	-.—				
M	--	Z	--..				

- Demo:
  - http://www.scphillips.com/morse/

Most frequent letters in English

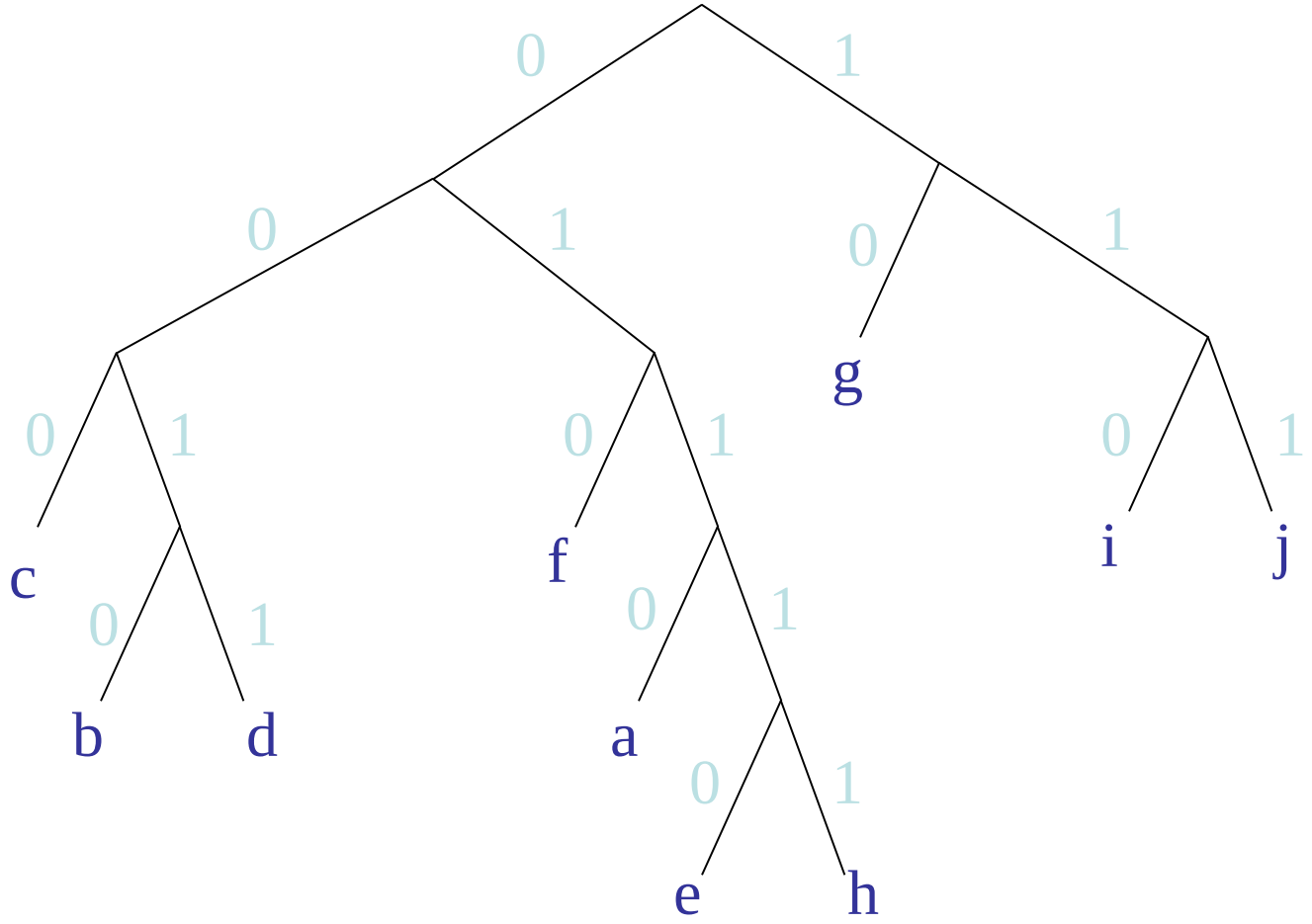
- Some are more frequently used than others…
- Most frequent letters:
  - E T A O I N S H R D L U
- Demo:
  - http://www.amstat.org/publications/jse/secure/v7n2/count-char.cfm
- Also: bigrams:
  - TH HE IN ER AN RE ND AT ON NT

Huffman coding

- Developed by David Huffman (1952)
- Average of 5 bits per character (37.5% compression)
- Based on frequency distributions of symbols
- Algorithm: iteratively build a tree of symbols starting with the two least frequent symbols

Symbol	*Frequency*
A	7
B	4
C	10
D	5
E	2
F	11
G	15
H	3
I	7
J	8

The Huffman tree



Symbol	*Code*
A	0110
B	0010
C	000
D	0011
E	01110
F	010
G	10
H	01111
I	110
J	111

POSTINGS COMPRESSION

Postings compression

- The postings file is much larger than the dictionary, factor of at least 10.
- Key: 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 ≈ 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 ~ 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 ~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 ~$\log_2 G$ 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 $c = 0$.

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 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

Unary code

- Represent *n* as *n* 1s with a final 0.
- Unary code for 3 is 1110.
- Unary code for 40 is

1111111111111111111111111111111111111110 .

- Unary code for 80 is:

11111111111111111111111111111111111111111111111111111111111111111111111110

- This doesn't look promising, but….

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 → 1101 → 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

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

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 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.

References

- *Introduction to Information Retrieval*, chapters 4 & 5.
- Some slides were adapted from
  - the book's companion website:
    - http://nlp.stanford.edu/IR-book/information-retrieval-book.html
  - Prof. Dragomir Radev's lectures at the University of Michigan:
    - http://clair.si.umich.edu/~radev/teaching.html