

Introduction to Information Retrieval

<http://informationretrieval.org>

IIR 5: Index Compression

Hinrich Schütze

Institute for Natural Language Processing, Universität Stuttgart

2009.05.11

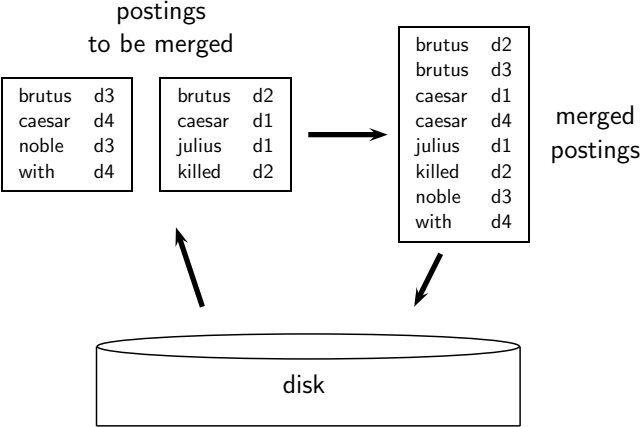
Overview

- 1 Recap
- 2 Term statistics
- 3 Dictionary compression
- 4 Postings compression

Outline

- 1 Recap
- 2 Term statistics
- 3 Dictionary compression
- 4 Postings compression

Blocked Sort-Based Indexing



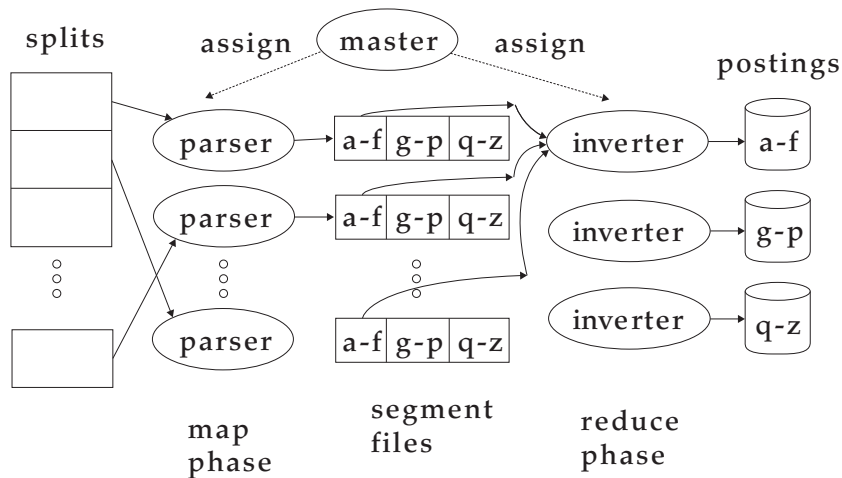
Single-pass in-memory indexing

- Abbreviation: SPIMI
- 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 algorithm

```
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
```

MapReduce for index construction



Dynamic indexing: Simplest approach

- Maintain **big main index on disk**
- New docs go into **small auxiliary index in memory**.
- Search across both, merge results
- Periodically, merge auxiliary index into big index
- Deletions:
 - Invalidation bit-vector for deleted docs
 - Filter docs returned by index using this bit-vector

Outline

- 1 Recap
- 2 Term statistics
- 3 Dictionary compression
- 4 Postings compression

Inverted index

For each term t , we store a list of all documents that contain t .

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

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

CALPURNIA → 2 | 31 | 54 | 101

⋮

⏟
dictionary

⏟
postings file

Today:

- How much space do we need for the dictionary?
- How much space do we need for the postings file?
- How can we compress them?

Why compression? (in general)

- Use less disk space (saves money)
- Keep more stuff in memory (increases speed)
- Increase speed of transferring data from disk to memory (again, increases speed)
 - [read compressed data and decompress]
is faster than
[read uncompressed data]
- Premise: Decompression algorithms are fast.
- This is true of the decompression algorithms we will use.

Why compression in information retrieval?

- First, we will consider space for dictionary
 - Main motivation for dictionary compression: make it small enough to keep in main memory
- Then for the postings file
 - Motivation: reduce disk space needed, decrease time needed to read from disk
 - Large search engines keep significant part of postings in memory
- We will devise various compression schemes.

Model collection: The Reuters collection

symbol	statistic	value
N	documents	800,000
L	avg. # word tokens per document	200
M	word types	400,000
	avg. # bytes per word token (incl. spaces/punct.)	6
	avg. # bytes per word token (without spaces/punct.)	4.5
	avg. # bytes per word type	7.5
T	non-positional postings	100,000,000

Effect of preprocessing for Reuters

size of	word types (term)			non-positional postings			positional postings (word tokens)		
	dictionary			non-positional index			positional index		
	size	Δ	cml..	size	Δ	cml..	size	Δ	cml..
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 w's	391,493	-0%	-19%	83,390,443	-14%	-24%	121,857,825	-31%	-38%
150 stop w's	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%

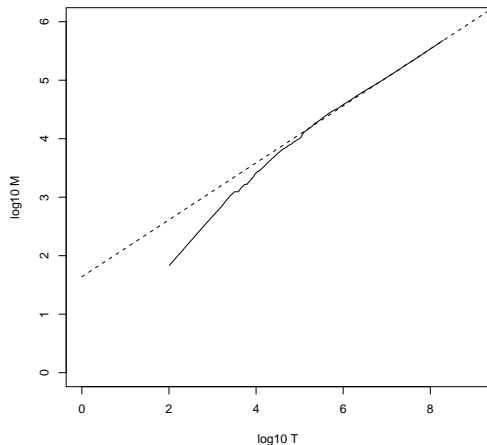
Lossy vs. lossless compression

- Lossy compression: Discard some information
- Several of the preprocessing steps we just saw can be viewed as lossy compression:
 - downcasing, stop words, porter, number elimination
- Lossless compression: All information is preserved.
 - What we mostly do in IR.

How big is the term vocabulary?

- That is, how many distinct words are there?
- Can we assume there is an upper bound?
- Not really: At least $70^{20} \approx 10^{37}$ different words of length 20.
- The vocabulary will keep growing with 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 for the parameters k and b are: $30 \leq k \leq 100$ and $b \approx 0.5$.
- Heaps' law is linear in log-log space.
 - It is the simplest possible relationship between collection size and vocabulary size in log-log space.
 - Empirical law

Heaps' law for Reuters



Vocabulary size M as a function of collection size T (number of tokens) for Reuters-RCV1. For these data, the dashed line $\log_{10} M = 0.49 * \log_{10} T + 1.64$ is the best least squares fit. Thus, $M = 10^{1.64} T^{0.49}$ and $k = 10^{1.64} \approx 44$ and $b = 0.49$.

Empirical fit for Reuters

- Good, as we just saw in the graph.
- Example: for the first 1,000,020 tokens Heaps' law predicts 38,323 terms:

$$44 \times 1,000,020^{0.49} \approx 38,323$$

- The actual number is 38,365 terms, very close to the prediction.
- Empirical observation: fit is good in general.

Excercise

- 1 What is the effect of including spelling errors vs. automatically correcting spelling errors on Heaps' law?
- 2 Compute vocabulary size M
 - 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 (2×10^{10}) pages, containing 200 tokens on average
 - What is the size of the vocabulary of the indexed collection as predicted by Heaps' law?

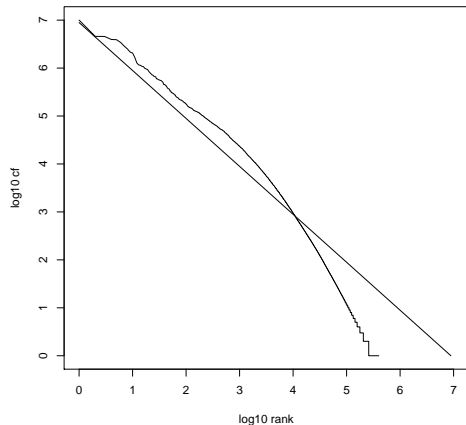
Zipf's law

- Now we have characterized the growth of the vocabulary in collections.
- We also want to know how many frequent vs. infrequent terms we should expect in a collection.
- 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 $1/i$.
- $cf_i \propto \frac{1}{i}$
- cf_i is collection frequency: the number of occurrences of the term t_i in the collection.

Zipf's law

- Zipf's law: The i^{th} most frequent term has frequency proportional to $1/i$.
- $cf_i \propto \frac{1}{i}$
- cf is collection frequency: the number of occurrences of the term in the collection.
- So if the most frequent term (*the*) occurs cf_1 times, then the second most frequent term (*of*) has half as many occurrences $cf_2 = \frac{1}{2}cf_1 \dots$
- \dots and the third most frequent term (*and*) has a third as many occurrences $cf_3 = \frac{1}{3}cf_1$ etc.
- Equivalent: $cf_i = ci^k$ and $\log cf_i = \log c + k \log i$ (for $k = -1$)
- Example of a power law

Zipf's law for Reuters



Fit is not great. What is important is the key insight: Few frequent terms, many rare terms.

Outline

- 1 Recap
- 2 Term statistics
- 3 Dictionary compression
- 4 Postings compression

Dictionary compression

- The dictionary is small compared to the postings file.
- But we want to keep it in memory.
- Also: competition with other applications, cell phones, onboard computers, fast startup time
- So compressing the dictionary is important.

Recall: Dictionary as array of fixed-width entries

term	document frequency	pointer to postings list
a	656,265	→
aachen	65	→
...
zulu	221	→

Space

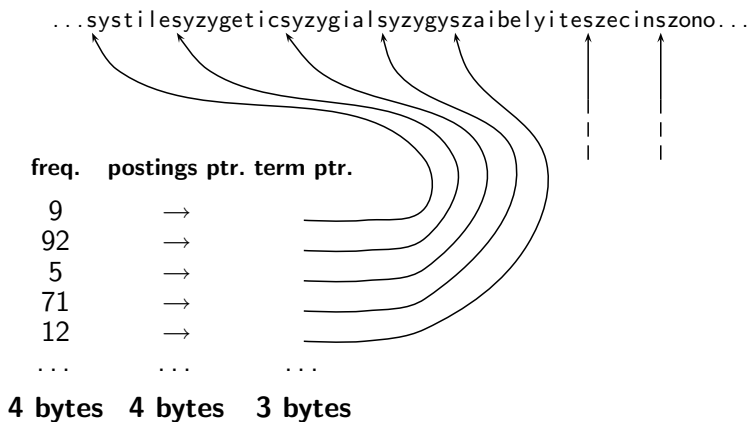
space needed: 20 bytes 4 bytes 4 bytes

for Reuters: $(20+4+4)*400,000 = 11.2$ MB

Fixed-width entries are bad.

- Most of the bytes in the term column are wasted.
 - We allot 20 bytes for terms of length 1.
- We can't handle HYDROCHLOROFLUOROCARBONS and SUPERCALIFRAGILISTICEXPIALIDOCIOUS
- Average length of a term in English: 8 characters
- How can we use on average 8 characters per term?

Dictionary as a string



Space for dictionary as a string

- 4 bytes per term for frequency
- 4 bytes per term for pointer to postings list
- 8 bytes (on average) for term in string
- 3 bytes per pointer into string (need $\log_2 8 \cdot 400000 < 24$ bits to resolve $8 \cdot 400,000$ positions)
- Space: $400,000 \times (4 + 4 + 3 + 8) = 7.6\text{MB}$ (compared to 11.2 MB for fixed-width array)

Dictionary as a string with blocking

...7systile9syzygetic8syzygial6syzygy11szaibelyite6szecin...

freq.	postings ptr.	term ptr.
-------	---------------	-----------

9	→	
---	---	--

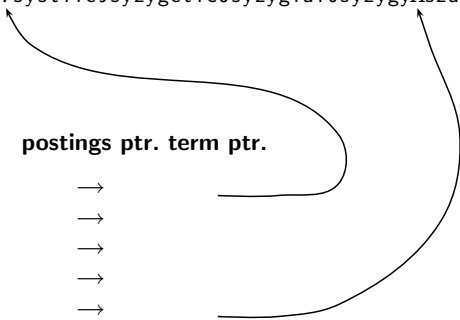
92	→	
----	---	--

5	→	
---	---	--

71	→	
----	---	--

12	→	
----	---	--

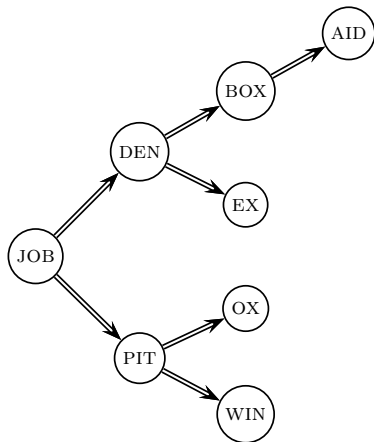
...
-----	-----	-----



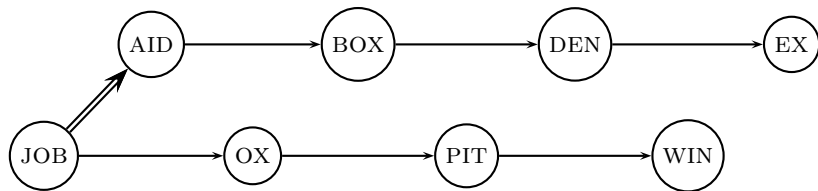
Space for dictionary as a string with blocking

- Example block size $k = 4$
- Where we used 4×3 bytes for term pointers without blocking
...
- ... we now use 3 bytes for one pointer plus 4 bytes for indicating the length of each term.
- We save $12 - (3 + 4) = 5$ bytes per block.
- Total savings: $400,000/4 * 5 = 0.5$ MB
- This reduces the size of the dictionary from 7.6 MB to 7.1 MB.

Lookup of a term without blocking



Lookup of a term with blocking: (slightly) slower



Front coding

One block in blocked compression ($k = 4$) ...

8 a u t o m a t a **8** a u t o m a t e **9** a u t o m a t i c **10** a u t o m a t i o n



... further compressed with front coding.

8 a u t o m a t * a **1** ◊ e **2** ◊ i c **3** ◊ i o n

Dictionary compression for Reuters: Summary

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

Outline

- 1 Recap
- 2 Term statistics
- 3 Dictionary compression
- 4 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 $\log_2 800,000 \approx 19.6 < 20$ bits per docID.
- Our goal: use a lot less than 20 bits per docID.

Key idea: Store gaps instead of docIDs

- Each postings list is ordered in increasing order of docID.
- Example postings list: COMPUTER: 283154, 283159, 283202, ...
- It suffices to store **gaps**: $283159-283154=5$, $283202-283154=43$
- Example postings list: COMPUTER: 283154, 5, 43, ...
- Gaps for frequent terms are small.
- Thus: We can encode small gaps with fewer than 20 bits.

Gap encoding

	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 and other rare terms, we will use about 20 bits per gap (= posting).
 - For THE and other very frequent terms, we will use only a few bits per gap (= posting).
- In order to implement this, we need to devise some form of **variable length encoding**.
- Variable length encoding uses few bits for small gaps and many bits for large gaps.

Variable byte (VB) code

- Used by many commercial/research systems
- Good low-tech blend of variable-length coding and sensitivity to alignment matches (bit-level codes, see later).
- Dedicate 1 bit (high bit) to be a **continuation bit** c .
- If the gap G fits within 7 bits, binary-encode it in the 7 available bits and set $c = 1$.
- Else: encode lower-order 7 bits and then use one or more 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 of the other bytes to 0 ($c = 0$).

VB code examples

docIDs	824		829		215406
gaps			5		214577
VB code	00000110	10111000	10000101	00001101	00001100 10110001

VB code encoding algorithm

VBENCODENUMBER(n)

```
1 bytes  $\leftarrow \langle \rangle$ 
2 while true
3 do PREPEND(bytes,  $n \bmod 128$ )
4   if  $n < 128$ 
5     then BREAK
6    $n \leftarrow n \text{ div } 128$ 
7 bytes[LENGTH(bytes)] += 128
8 return bytes
```

VBENCODE($numbers$)

```
1 bytestream  $\leftarrow \langle \rangle$ 
2 for each  $n \in numbers$ 
3 do bytes  $\leftarrow$  VBENCODENUMBER( $n$ )
4   bytestream  $\leftarrow$  EXTEND(bytestream, bytes)
5 return bytestream
```

VB code decoding algorithm

VBDECODE(*bytestream*)

1 *numbers* $\leftarrow \langle \rangle$

2 $n \leftarrow 0$

3 **for** $i \leftarrow 1$ **to** LENGTH(*bytestream*)

4 **do if** *bytestream*[i] < 128

5 **then** $n \leftarrow 128 \times n + \textit{bytestream}[i]$

6 **else** $n \leftarrow 128 \times n + (\textit{bytestream}[i] - 128)$

7 APPEND(*numbers*, n)

8 $n \leftarrow 0$

9 **return** *numbers*

Other variable codes

- Instead of bytes, we can also use a different “unit of alignment”: 32 bits (words), 16 bits, 4 bits (nibbles) etc
- Variable byte alignment wastes space if you have many small gaps – nibbles do better on those.
- Recent work on word-aligned codes that efficiently “pack” a variable number of gaps into one word.
- See resources at the end

Gamma code

- Represent a gap G as a pair of **length** and **offset**.
- Offset is the gap in binary, with the leading bit chopped off.
- For example $13 \rightarrow 1101 \rightarrow 101 = \text{offset}$
- Length is the length of offset.
- For 13 (offset 101), this is 3.
- Encode length in **unary** code: 1110.
- Gamma code of 13 is the concatenation of length and offset: 1110101.

Gamma code examples

number	unary code	length	offset	γ code
0	0			
1	10	0		0
2	110	10	0	10,0
3	1110	10	1	10,1
4	11110	110	00	110,00
9	111111110	1110	001	1110,001
13		1110	101	1110,101
24		11110	1000	11110,1000
511		111111110	11111111	111111110,11111111
1025		11111111110	0000000001	11111111110,0000000001

Exercise

- Compute the variable byte code of 130
- Compute the gamma code of 130

Length of gamma code

- The length of *offset* is $\lfloor \log_2 G \rfloor$ bits.
- The length of *length* is $\lfloor \log_2 G \rfloor + 1$ bits,
- So the length of the entire code is $2 \times \lfloor \log_2 G \rfloor + 1$ bits.
- γ codes are always of odd length.
- Gamma codes are within a factor of 2 of the optimal encoding length $\log_2 G$.
 - (assuming the frequency of a gap G is proportional to $\log_2 G$ – not really true)

Gamma code: Properties

- Gamma code is **prefix-free**.
- Encoding is optimal within a factor of 3 (and within a factor of 2 making additional assumptions).
- This result is independent of the distribution of gaps!
- We can use gamma codes for any distribution. Gamma code is **universal**.
- Gamma code is **parameter-free**.

Gamma codes: Alignment

- Machines have word boundaries – 8, 16, 32 bits
- Compressing and manipulating at 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.

Term-document incidence matrix

	Anthony and Cleopatra	Julius Caesar	The Tempest	Hamlet	Othello	Macbeth	...
ANTHONY	1	1	0	0	0	1	
BRUTUS	1	1	0	1	0	0	
CAESAR	1	1	0	1	1	1	
CALPURNIA	0	1	0	0	0	0	
CLEOPATRA	1	0	0	0	0	0	
MERCY	1	0	1	1	1	1	
WORSER	1	0	1	1	1	0	

...

Entry is 1 if term occurs. Example: CALPURNIA occurs in *Julius Caesar*. Entry is 0 if term doesn't occur. Example: CALPURNIA doesn't occur in *The tempest*.

Compression of Reuters

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)	3600.0
collection (text)	960.0
T/D 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

Summary

- We can now create an index for highly efficient Boolean retrieval that is very space efficient.
- Only 10-15% of the total size of the text in the collection.
- However, we've ignored positional and frequency information.
- For this reason, space savings are less in reality.

Resources

- Chapter 5 of IIR
- Resources at <http://ifnlp.org/ir>
 - Original publication on word-aligned binary codes by Anh and Moffat (2005)
 - Original publication on variable byte codes by Scholer, Williams, Yiannis and Zobel (2002)
 - More details on compression (including compression of positions and frequencies) in Zobel and Moffat (2006)