Introduction to Information Retrieval and Text Mining Index Compression

Roman Klinger

Institute for Natural Language Processing, University of Stuttgart

2021-11-09

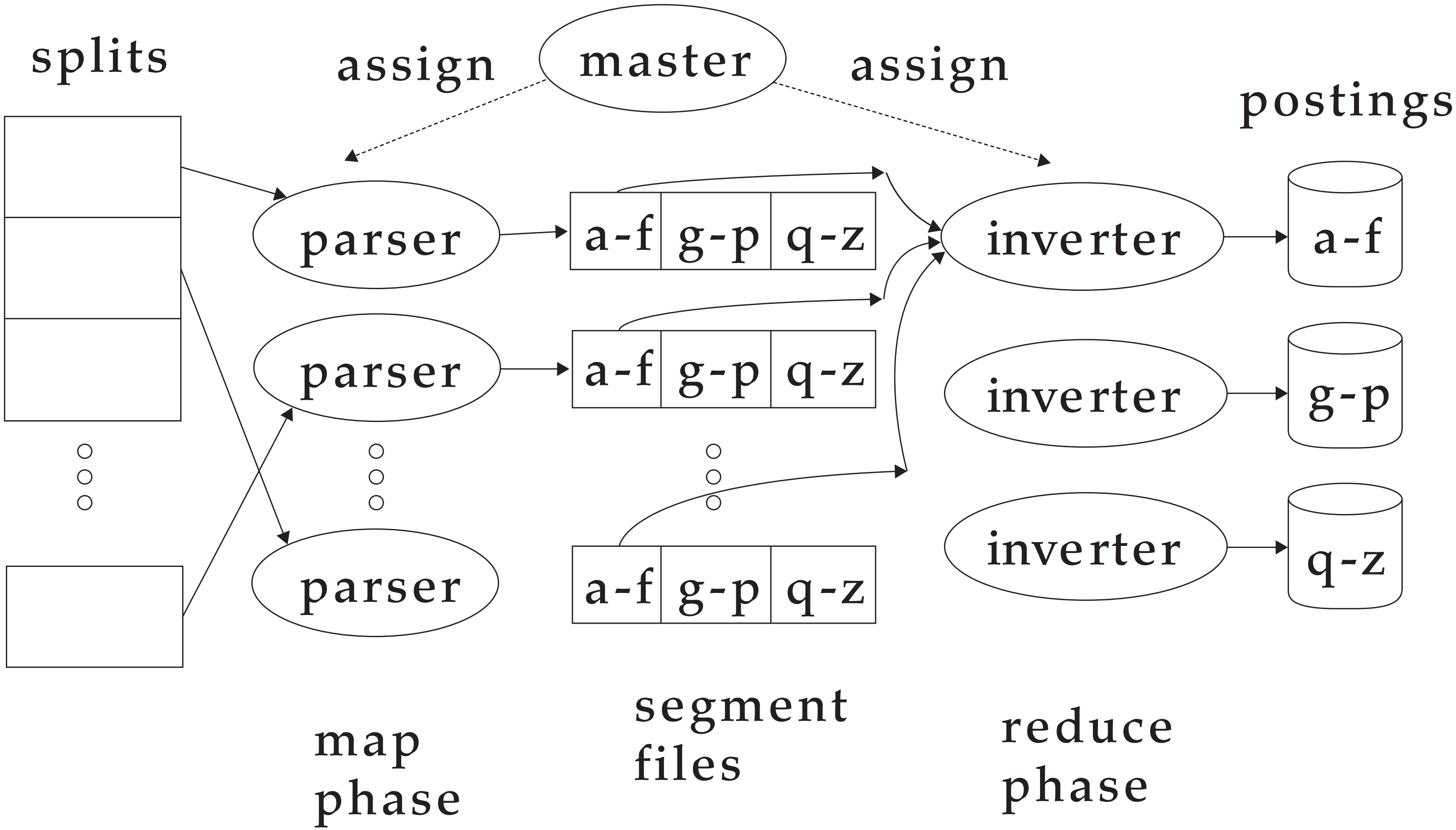
Overview

Outline

Recap: Indexing Algorithms

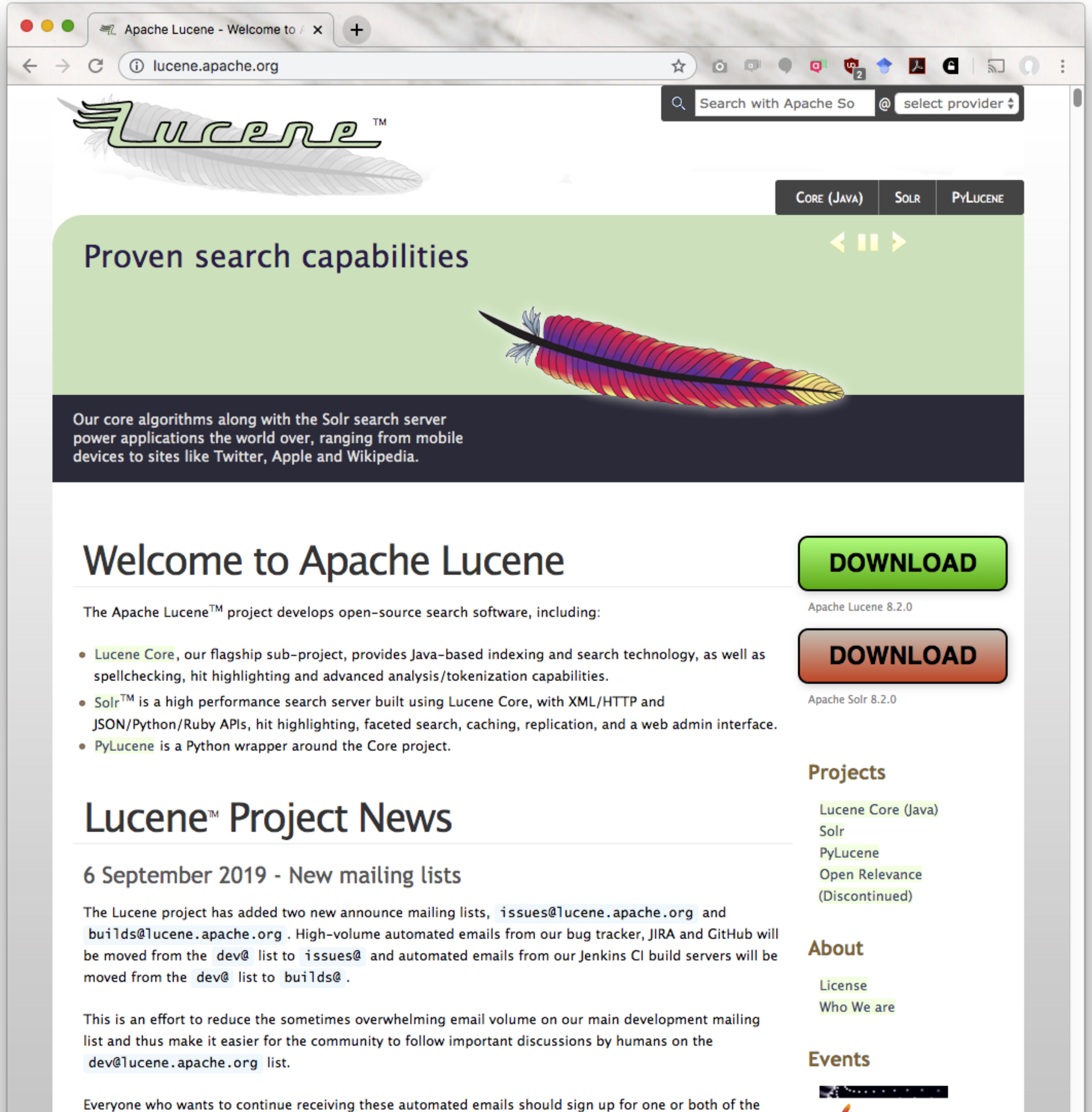
- Blocked Sort-Based Indexing (BSBI)
- Single-pass in-memory indexing (SPIMI)
  - Both build blocks of documents
  - BSBI keeps one dictionary and sorts each block into it
  - SPIMI build full indexes for each block and merges later

Recap: MapReduce for index construction



Recap: Dynamic indexing and Logarithmic Merging

- 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
- More efficient (but same idea): Logarithmic Merging


Apache Lucene - Welcome to
lucene.apache.org
Search with Apache So
select provider
Lucene™
Core (Java)
Solr
PyLucene
Proven search capabilities
Our core algorithms along with the Solr search server power applications the world over, ranging from mobile devices to sites like Twitter, Apple and Wikipedia.
Welcome to Apache Lucene
DOWNLOAD
Apache Lucene 8.2.0
DOWNLOAD
Apache Solr 8.2.0
The Apache Lucene™ project develops open-source search software, including:
Lucene Core, our flagship sub-project, provides Java-based indexing and search technology, as well as spellchecking, hit highlighting and advanced analysis/tokenization capabilities.
Solr™ is a high performance search server built using Lucene Core, with XML/HTTP and JSON/Python/Ruby APIs, hit highlighting, faceted search, caching, replication, and a web admin interface.
PyLucene is a Python wrapper around the Core project.
Projects
Lucene Core (Java)
Solr
PyLucene
Open Relevance (Discontinued)
Lucene™ Project News
6 September 2019 - New mailing lists
The Lucene project has added two new announce mailing lists, issues@lucene.apache.org and builds@lucene.apache.org. High-volume automated emails from our bug tracker, JIRA and GitHub will be moved from the dev@ list to issues@ and automated emails from our Jenkins CI build servers will be moved from the dev@ list to builds@.
About
License
Who We are
This is an effort to reduce the sometimes overwhelming email volume on our main development mailing list and thus make it easier for the community to follow important discussions by humans on the dev@lucene.apache.org list.
Events
Everyone who wants to continue receiving these automated emails should sign up for one or both of the

IRTM 21/22 Schedule

	Date	Session	TOPIC	Assignments	Group
DO	21.10.2021	1	Introduction and Boolean Retrieval		Gerade 2
DI	26.10.2021	2	Term Vocabularies and Postings Lists		Ungerade 1
DO	28.10.2021	3	Dictionaries, Phrase Queries		Ungerade 2
DI	02.11.2021	4	Spelling, Tolerant Retrieval	Publication Assignment 1	Gerade 1
DO	04.11.2021	5	Index Construction		Gerade 2
DI	09.11.2021	6	Compression	Deadline Assignment 1	Ungerade 1
DO	11.11.2021	7	Scoring	Publication Assignment 2	Ungerade 2
DI	16.11.2021		DISCUSSION ASSIGMENT 1		Online Only
DO	18.11.2021	8	Ranking		Gerade 2
DI	23.11.2021	9	System, Summaries, Intro to Evaluation		Ungerade 1
DO	25.11.2021	10	Evaluation, IA Agreement	Deadline Assignment 2	Ungerade 2
DI	30.11.2021	11	Query Expansion, Probabilistic Retrieval, Lang Models	Publication Assignment 3	Gerade 1
DO	02.12.2021		DISCUSSION ASSIGNMENT 2		Online Only
DI	7.12.2021	12	LM, Text Classification		Ungerade 1
DO	09.12.2021	13	TC, NB		Ungerade 2
DI	14.12.2021	14	NB, Evaluation, MaxEnt	Deadline Assignment 3	Gerade 1
DO	16.12.2021	15	Feature Selection, Vector Space Classification, Perceptron	Publication Assignment 4	Gerade 2
DI	21.12.2021		DISCUSSION ASSIGNMENT 3		Online Only
DI	11.01.2022	16	Support Vector Machines, Learning to Rank		Gerade 1
DO	13.01.2022	17	Representation Learning and Deep Learning for TC		Gerade 2
DI	18.01.2022	18	Introduction to Clustering	Deadline Assignment 4	Ungerade 1

Attendance in Lecture Hall

	Anmerkung				
Standardgruppe		/		/	135
Gerade: Even weeks	50	/	50	/	28
Lehrveranstaltungsaufzeichnung	0	/	0	/	0
	Gruppe dient ausschlie Anmeldung erfolgt nur a				
Ungerade: Odd weeks	50	/	50	/	18

I will put Gerade/Ungerade together next Monday, if nothing surprising happens.

Take-away today

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**								

- Motivation for compression in information retrieval systems
- How can we compress the dictionary component of the inverted index?
- How can we compress the postings component of the inverted index?
- Term statistics:
  how are terms distributed in document collections?
  (and how can we estimate these numbers)

Outline

Inverted index

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

BRUTUS	$\longrightarrow$	1	2	4	11	31	45	173	174	
CAESAR	$\longrightarrow$	1	2	4	5	6	16	57	132	...
CALPURNIA	$\longrightarrow$	2	31	54	101					
⋮										

dictionary **postings file**

Next:

- 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 (increases speed)
  - [read compressed data and decompress in memory] is faster than [read uncompressed data]
- Premise: Decompression algorithms are fast.

  (True for those studied here.)

Why compression in information retrieval?

- Consider space for dictionary
  - Main motivation for dictionary compression: make it small enough to keep in main memory
- Postings file
  - Motivation:
    reduce disk space needed,
    decrease time needed to read from disk

Lossy vs. lossless compression

- Lossy compression: Discard some information
    - Famous examples: MP3 (audio), JPEG (photos)
    - Some preprocessing steps can be viewed as lossy compression:
        - downcasing, stop words, stemming, number elimination
- Lossless compression: All information is preserved.
    - What we mostly do in index compression

Outline

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 type	7.5
T	non-positional postings	100,000,000

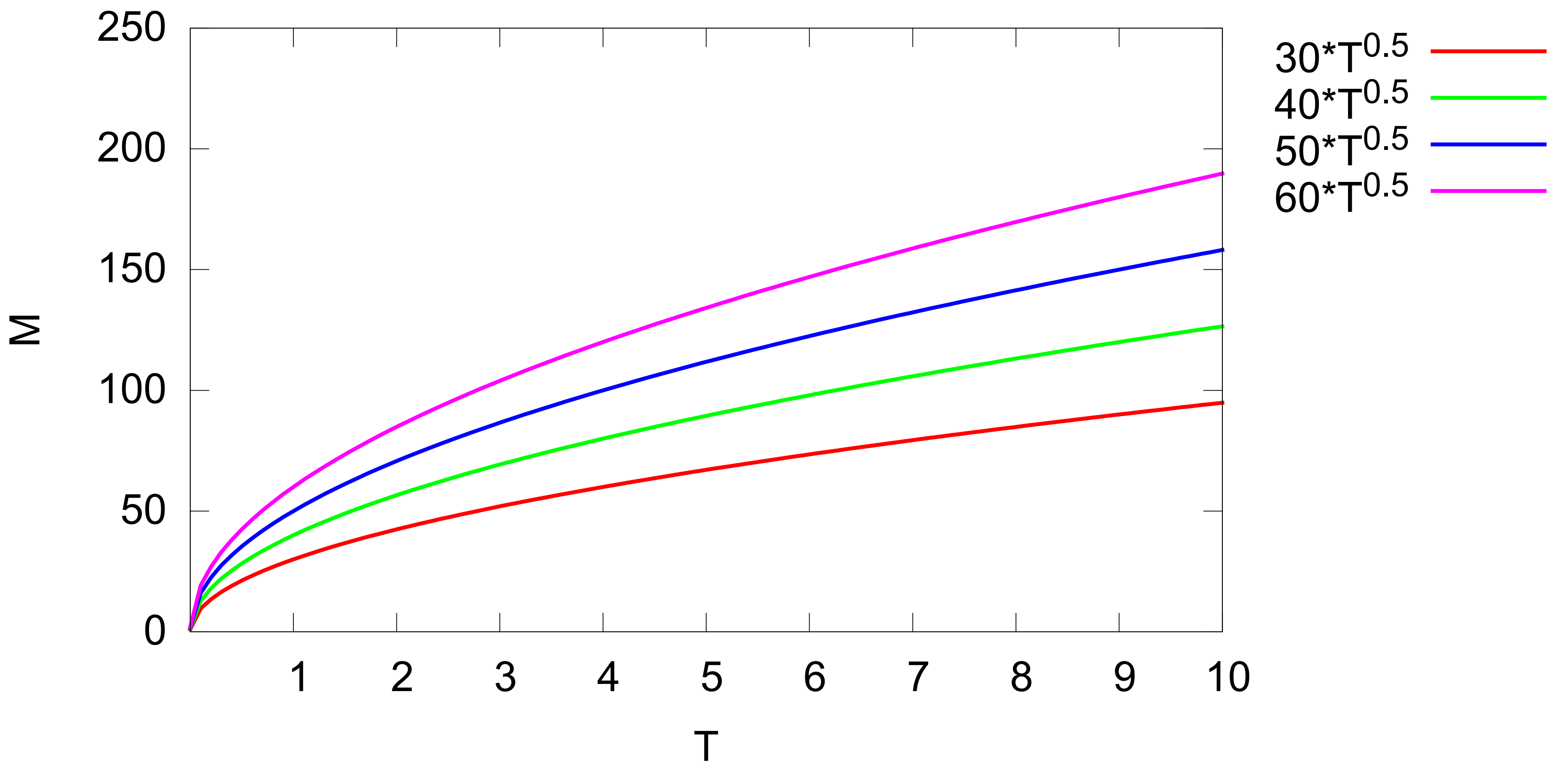
Effect of preprocessing for Reuters

size of	word types (terms)			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 stopw's	391,493	-0	-19	83,390,443	-14	-24	121,857,825	-31	-38
150 stopw'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

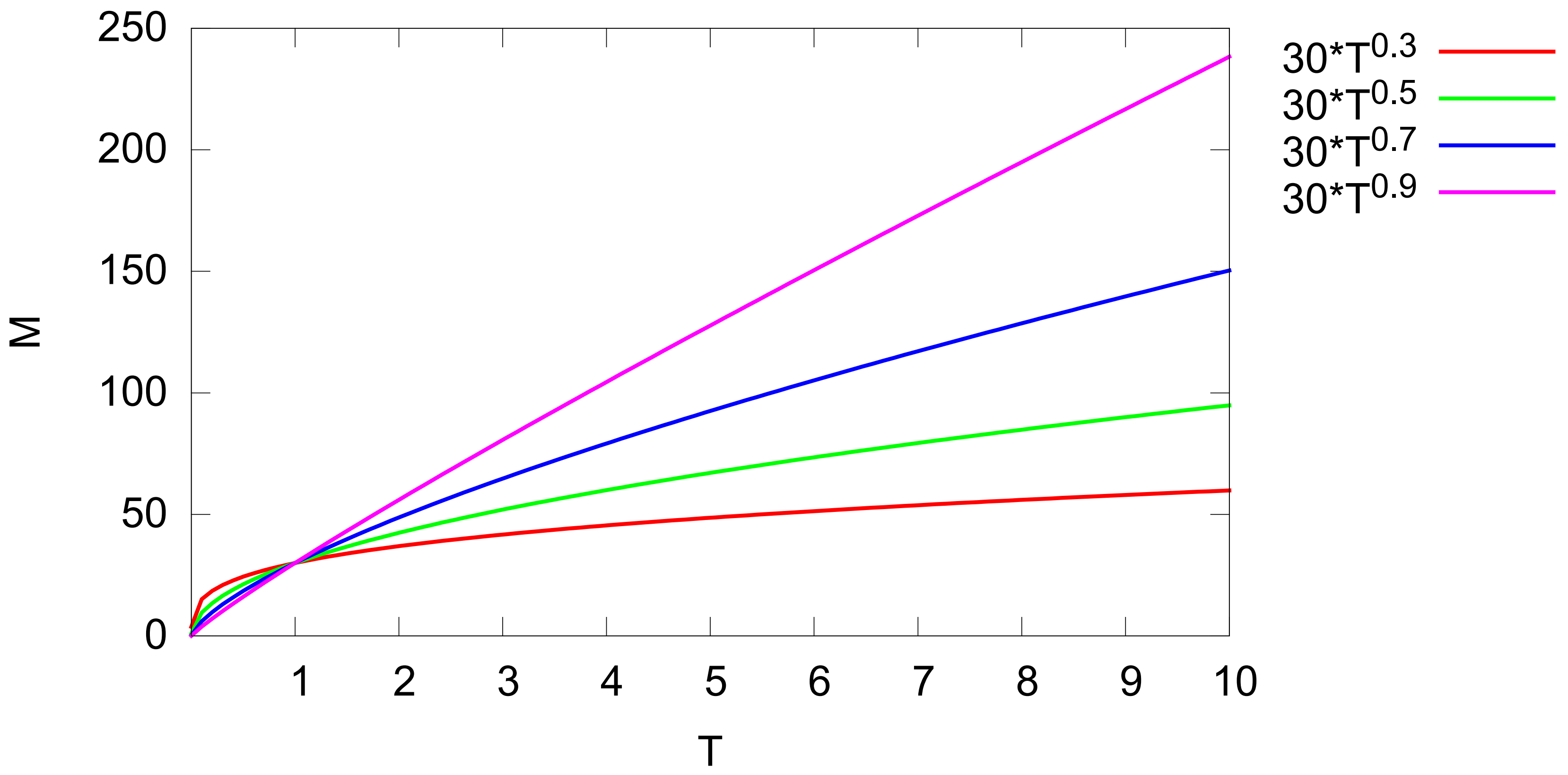
How big is the term vocabulary?

- That is, how many distinct words are there?
- Can we assume there is an upper bound??
  - Not really: Many long words, very infrequent..
  - The vocabulary will keep growing with collection size. But how much
- Heaps' law: $M = kT^b$
  - M is the size of the vocabulary (terms) 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.
- Empirical law

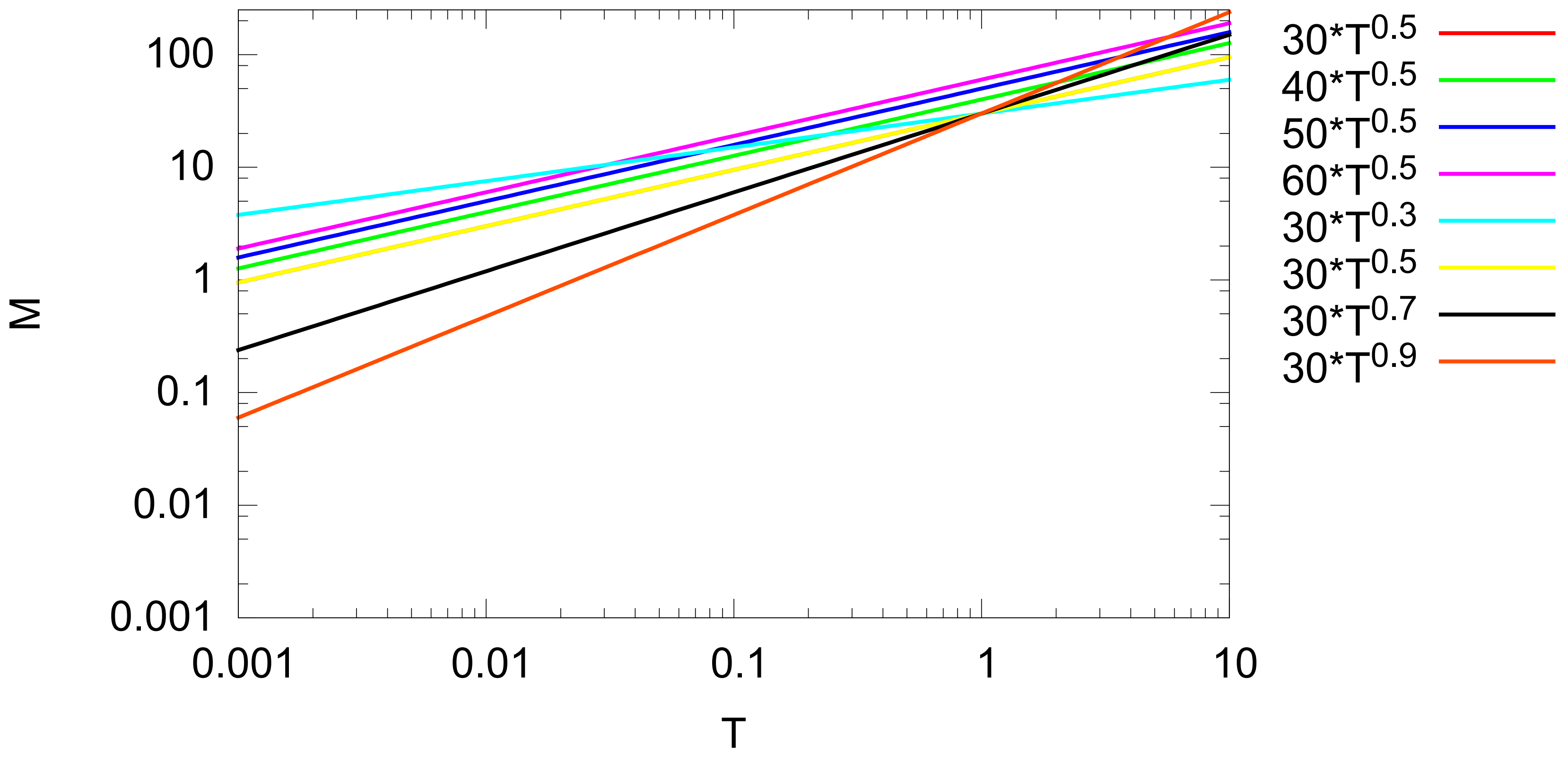
Examples for Heaps' Law (change k)



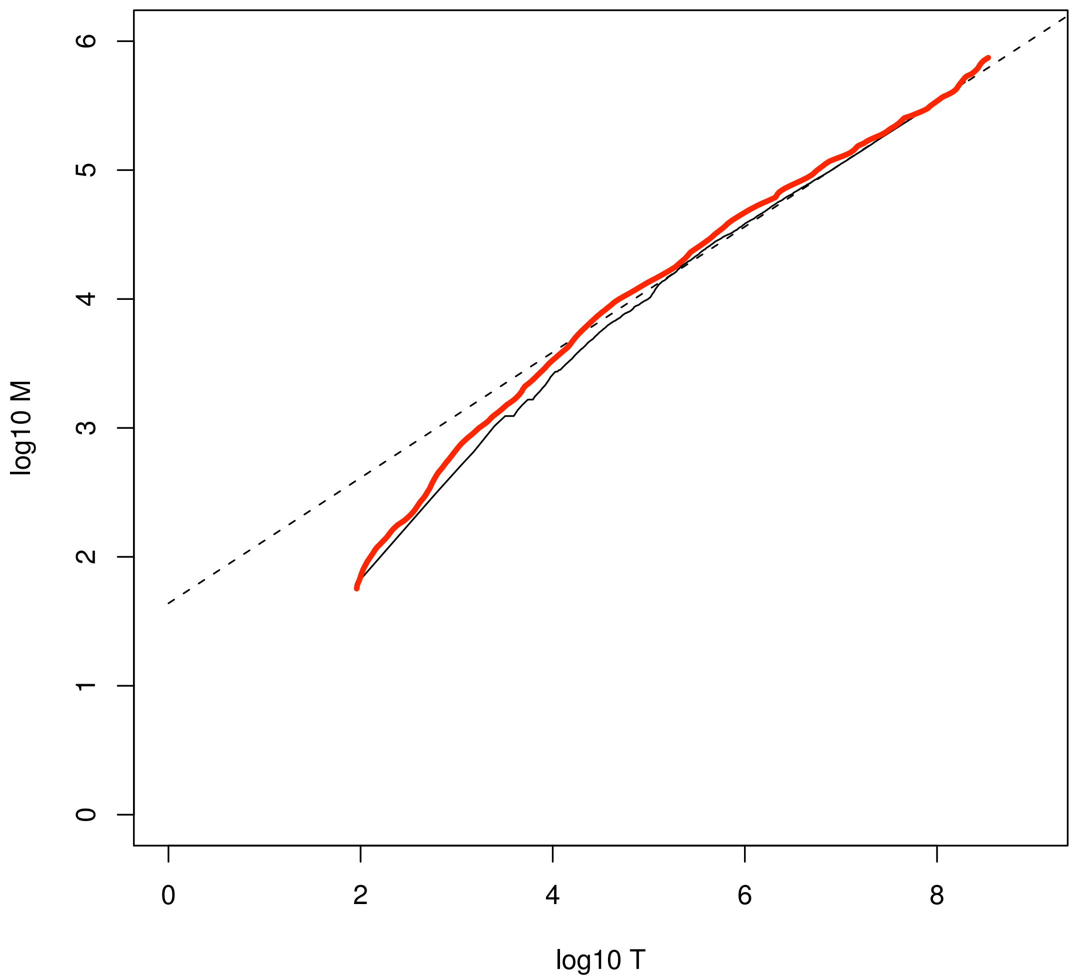
Examples for Heaps' Law (change b)



Examples for Heaps' Law (in log-log)



Heaps' law for Reuters



Reuters-RCV1

- $M = kT^b$
- Vocabulary Size M
- Num. Tokens T
- Dashed line: $\log_{10} M = 0.49 * \log_{10} T + 1.64$
- $M = 10^{1.64} T^{0.49}$
- $k = 10^{1.64} \approx 44$
- $b = 0.49$

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.

Exercise

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?

Reminder: $M = kT^b$

Exercise

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?

Reminder: $M = kT^b$

Idea:

- $\underbrace{3000}_{\text{Terms}} = k \cdot \underbrace{10000^b}_{\text{Tokens}}$ and $\underbrace{30000}_{\text{Terms}} = k \cdot \underbrace{1000000^b}_{\text{Tokens}}$

Solve Exercise

- $3000 = k \cdot 10000^b$ and $30000 = k \cdot 1000000^b$
- $b = 0.5$ and $k = 30$

$\Rightarrow$ $M = 30T^{0.5}$

- Given $T = 4,000,000,000,000 = 200 \cdot 20,000,000,000$:
  $M = 30 \cdot 4,000,000,000,000^{0.5}$
  - $M = 30 \cdot 2,000,000$

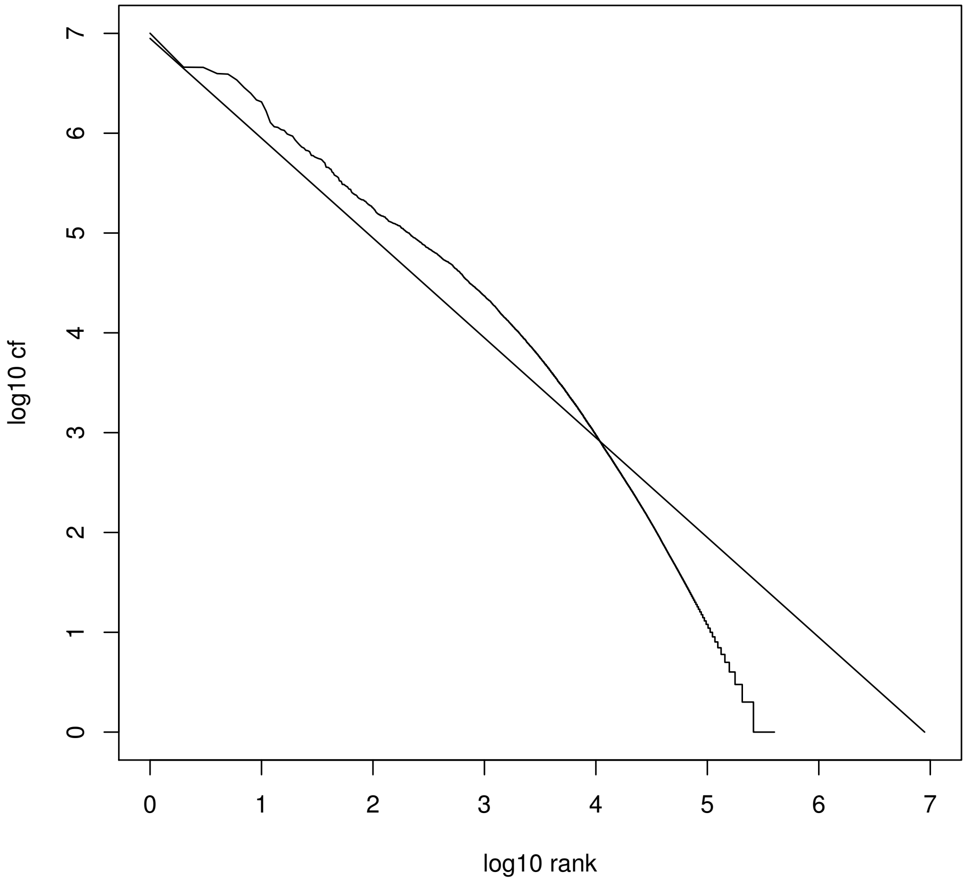
Zipf's law

- Heaps' Law: growth of the vocabulary in collections.
- What about frequent vs. infrequent terms to be expected in a collection?
- In natural language, there are a few very frequent terms and very many very rare terms.
- Zipf's law:
  i^{th} most frequent term has frequency cf_i proportional to $1/i$.
  - $\mathrm{cf}_i \propto \frac{1}{i}$
  - cf_i is collection frequency:
    the number of occurrences of the term t_i in the collection.

Zipf's law

- Zipf's law:
  i^{th} most frequent term has frequency cf_i proportional to $1/i$.
  - $\mathrm{cf}_i \propto \frac{1}{i}$
  - cf_i is collection frequency:
    the number of occurrences of the term t_i in the collection.
- In words:
  if the most frequent term (*the*) occurs cf_1 times
  the second most frequent term (*of*) has half as many occurrences $\mathrm{cf}_2 = \frac{1}{2}\mathrm{cf}_1 \ldots$
- ...and the third most frequent term (*and*) has a third as many occurrences $\mathrm{cf}_3 = \frac{1}{3}\mathrm{cf}_1$ etc.
- Example of a power law

Zipf's law for Reuters



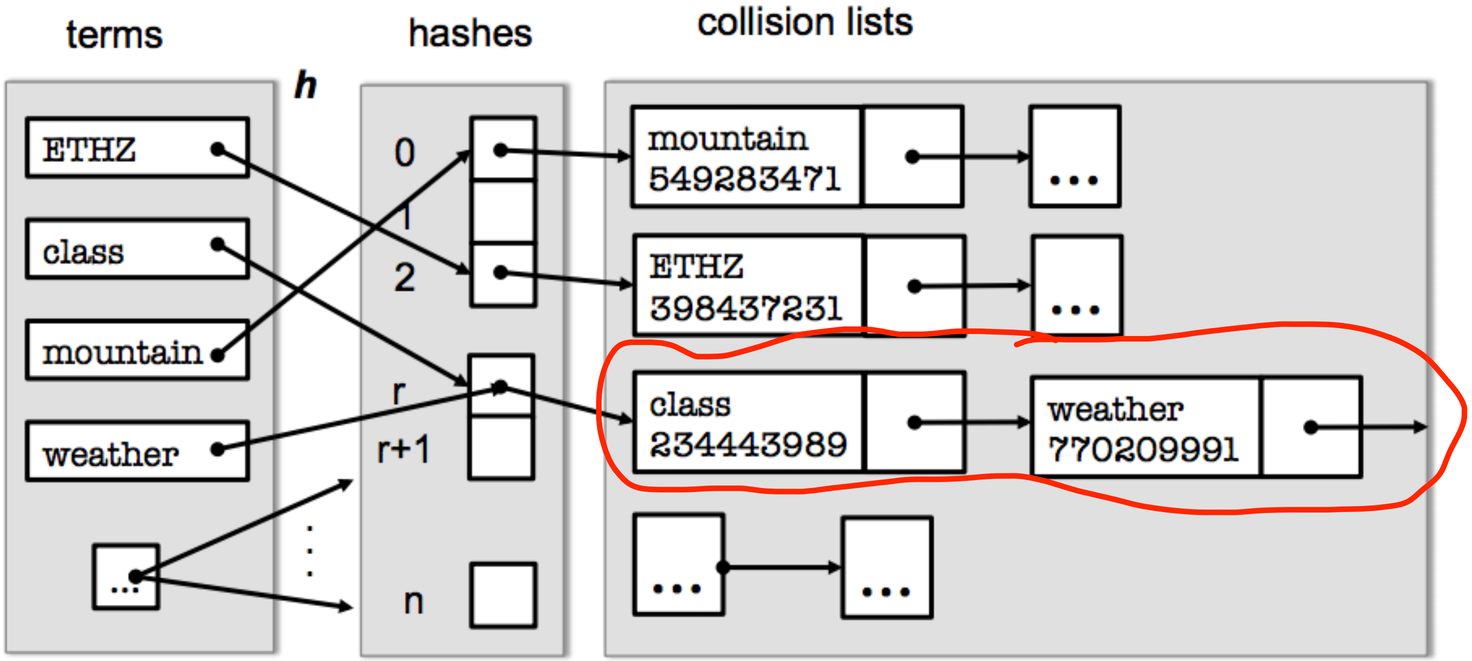
Fit is not great. What is important is the key insight: Few frequent terms, many rare terms.

Outline

Dictionary compression

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

Dictionary Terms and a Hash



Fixed (known) function

Storage need for token strings

Recall: Dictionary as array of fixed-width entries

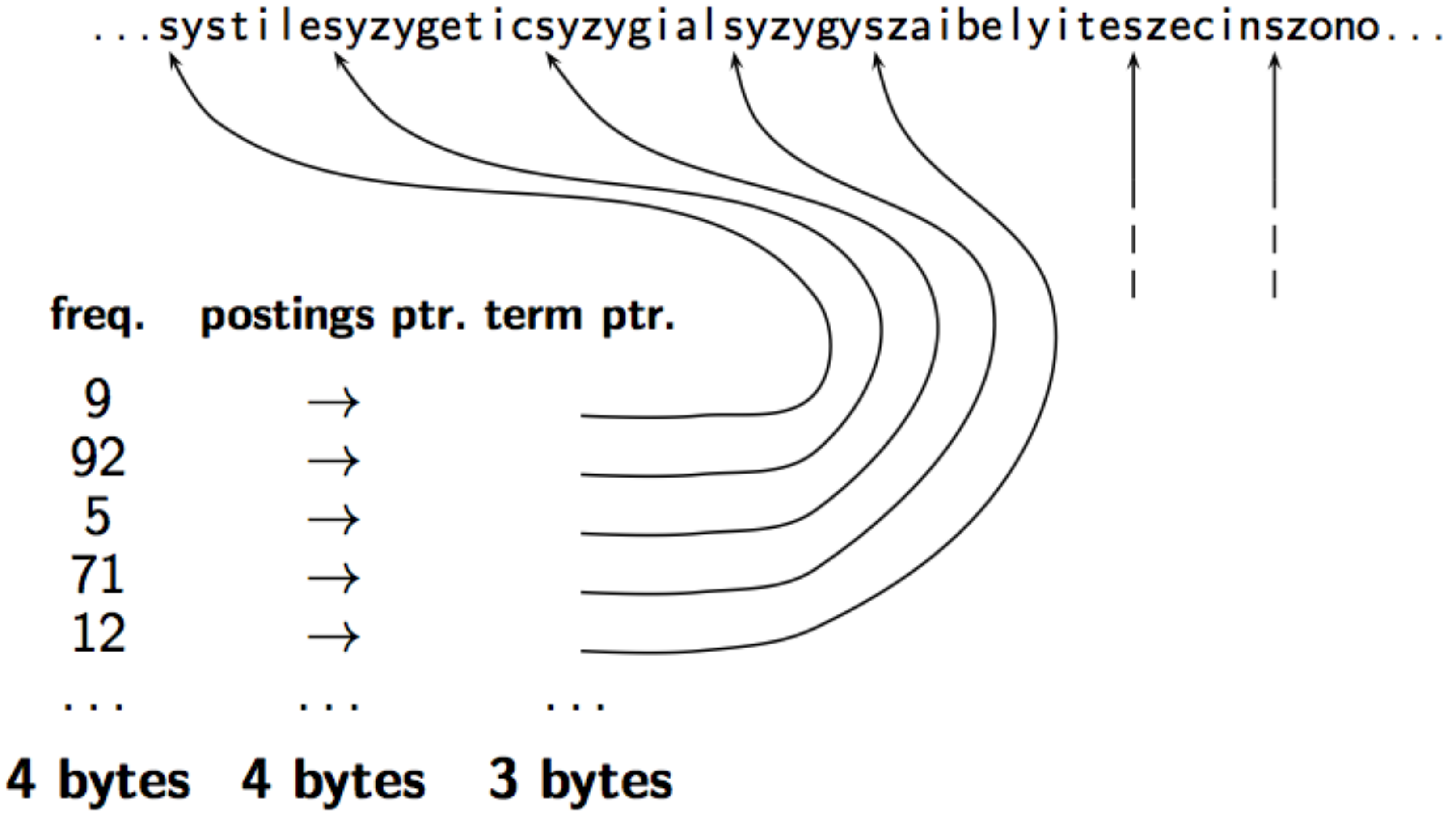
	term	document frequency	pointer to postings list
	a	656,265	$\longrightarrow$
	aachen	65	$\longrightarrow$
	...	...	...
	zulu	221	$\longrightarrow$
space needed:	20 bytes	4 bytes	4 bytes

Space 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 (or a little bit less)
- 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.

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

$\Downarrow$

...further compressed with front coding.

8 a u t o m a t $*$ a **1** $\diamond$ e **2** $\diamond$ i c **3** $\diamond$ 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
$\sim$, with blocking, $k = 4$	7.1
$\sim$, with blocking & front coding	5.9

Question

- Which prefixes should be used for front coding? What are the tradeoffs?
- Input: list of terms (= the term vocabulary)
- Output: list of prefixes that will be used in front coding

Outline

Postings compression

- The postings file is much larger than the dictionary, factor of at least 10.
- Key idea: store each posting compactly
- A posting for our purposes is a docID.
  (procedure can be used analogously for more entries)

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,
  283202-283159=43
- Example postings list using gaps:
  COMPUTER: 283154, 5, 43, ...
- Gaps for frequent terms are small.

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 do 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

- Key idea:
  - Dedicate 1 bit (high bit) to be a continuation bit c.
    0000 0000
- If the gap G fits within 7 bits:
  binary-encode it in the 7 available bits and set $c = 1$.
  e.g. if the gap is $127_{10} = 01111111_2$ result is: $\boxed{11111111}$
- Else (e.g. with $128_{10} = 10000000_2$): 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 ($c = 0$).
  - e.g. $128_{10} = 10000000_2$: 00000001 10000000

cont. bit

VB code examples

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

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.

Gamma codes for gap encoding

- You can get even more compression with another type of variable length encoding: bitlevel code.
- Gamma code is the best known of these.
- Introduce unary code first
- Unary code
  - Represent n as n 1s with a final 0.
  - Unary code for 3 is 1110
  - Unary code for 40 is 11111111111111111111111111111111111111110
  - Unary code for 70 is:

11111111111111111111111111111111111111111111111111111111111111111111110

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 $13_{10} \rightarrow 1101_2 \rightarrow 101 =$ offset
- Length is the length of offset.
- For 13_{10} (offset 101), this is 3.
- Encode length in unary code: 1110_1
- Gamma code of 13_{10} 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	1111111110	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

$(130_{10} = 10000010_2)$

Exercise solution

- $130_{10} = 10000010_2$
- In VB code: 0000 0001 1000 0010
- In Gamma Code:
  - Offset 000 0010
  - Length $7_{10} = 11111110_1$
  - Gamma code: 11111110 0000010

Length of gamma code

- The length of *offset* is $\lfloor \log_2 G \rfloor$ bits.
  - (number of bits used to represent a decimal number in general: $\lfloor \log_2(n) \rfloor + 1$)
- 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 (approximately) within a factor of 2 of the optimal encoding length $\log_2 G$.

Gamma code: Properties

- Gamma code is prefix-free: a valid code word is not a prefix of any other valid code.
- 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.

Compression of Reuters

data structure	size in MB
dictionary, fixed-width	11.2
dictionary, term pointers into string	7.6
$\sim$, with blocking, $k = 4$	7.1
$\sim$, 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

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
$\sim$, with blocking, $k = 4$	7.1
$\sim$, 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

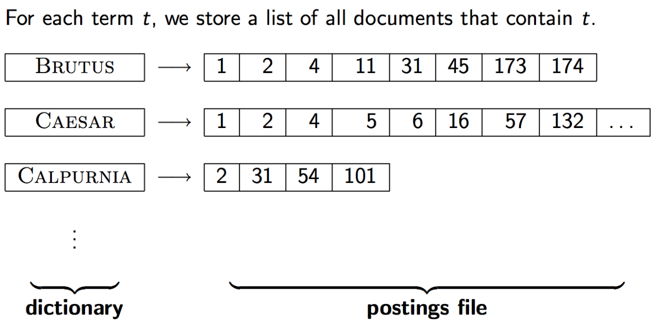
Compression of Reuters

data structure	size in MB
dictionary, fixed-width	11.2
dictionary, term pointers into string	7.6
$\sim$, with blocking, $k = 4$	7.1
$\sim$, 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 for Compression

- 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.

Take-away today



- Motivation for compression in information retrieval systems
- How can we compress the dictionary component of the inverted index?
- How can we compress the postings component of the inverted index?
- Term statistics:
  how are terms distributed in document collections?
  (and how can we estimate these numbers)