Introduction to Information Retrieval
(Manning, Raghavan, Schutze)

Chapter 3

Dictionaries and Tolerant retrieval

Chapter 4

Index construction

Chapter 5

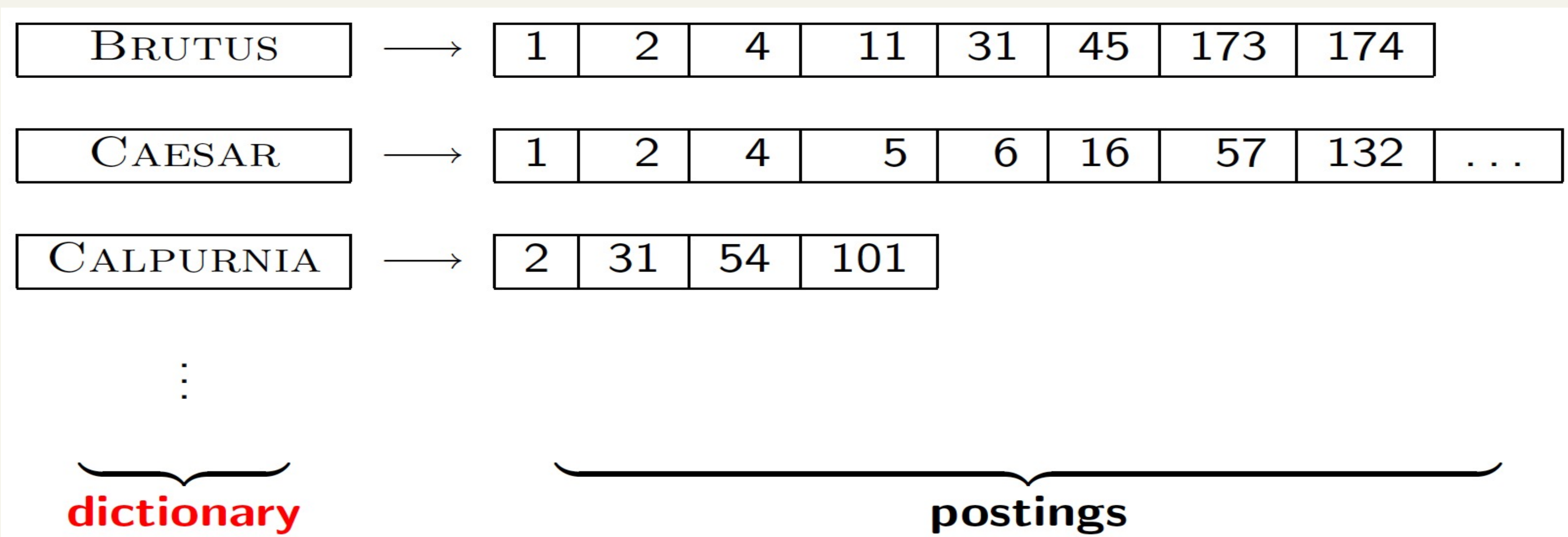
Index compression

Content

- Dictionary data structures
- “Tolerant” retrieval
  - Wild-card queries
  - Spelling correction
  - Soundex

Dictionary

- The dictionary is the data structure for storing the term vocabulary
- For each term, we need to store:
  - document frequency
  - pointers to each postings list



Dictionary data structures

- Two main choices:
  - Hash table
  - Tree
- Some IR systems use hashes, some trees
- Criteria in choosing hash or tree
  - fixed number of terms or keep growing
  - Relative frequencies with which various keys are accessed
  - How many terms

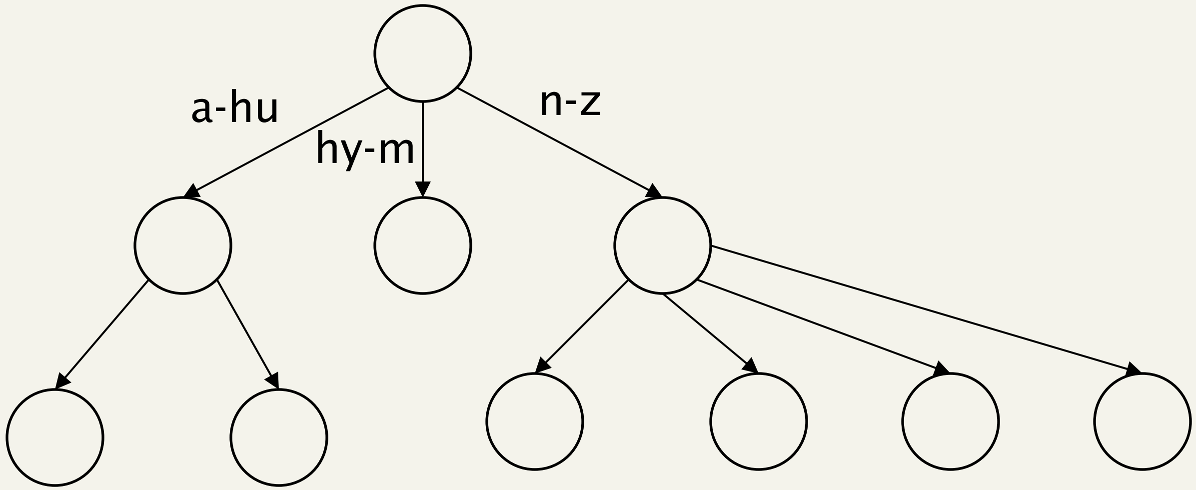
Hashes

- Each vocabulary term is hashed to an integer
  - (We assume you've seen hashtables before)
- Pros:
  - Lookup is faster than for a tree: O(1)
- Cons:
  - No easy way to find minor variants:
    - judgment /judgement
  - No prefix search
    - all terms starting with automat
  - Need to rehash everything periodically if vocabulary keeps growing

Trees

- Simplest: binary tree
- More usual: B-trees
- Pros:
  - Solves the prefix problem (finding all terms starting with *automat*)
- Cons:
  - Slower: O(log *M*) [and this requires *balanced* tree]
  - Rebalancing binary trees is expensive
    - But B-trees mitigate the rebalancing problem

B-tree



- Definition: Every internal nodel has a number of children in the interval [*a*,*b*] where *a, b* are appropriate natural numbers, e.g., [2,4].

Bla…

- Wild-card queries
  - ***mon*:*** find all docs containing any word beginning "mon".

- Spell correction
- Document correction

- Use different forms of inverted indexes
  - Standard inverted index (chapters 1 &2)
  - Permuterm index
  - *k*-gram indexes

Chapter 3

Dictionaries and Tolerant retrieval

Chapter 4

Index construction

Chapter 5

Index compression

Index construction

- How do we construct an index?
- What strategies can we use with limited main memory?
- Many design decisions in information retrieval are based on the characteristics of hardware ...
- Scaling index construction

RCV1: our corpus

- Shakespeare's collected works definitely aren't large enough for demonstrating many of the points in this course.
- The corpus 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
	avg. # bytes per term	7.5
	non-positional postings	100,000,000

4.5 bytes per word token vs. 7.5 bytes per word type: why?

Construction algorithms

- BSBI: Blocked sort-based indexing
- SPIMI: Single-pass in-memory indexing

Distributed indexing

- For web-scale indexing (don't try this at home!):
  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 use 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 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 in a pool.

Parallel tasks

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

Parsers

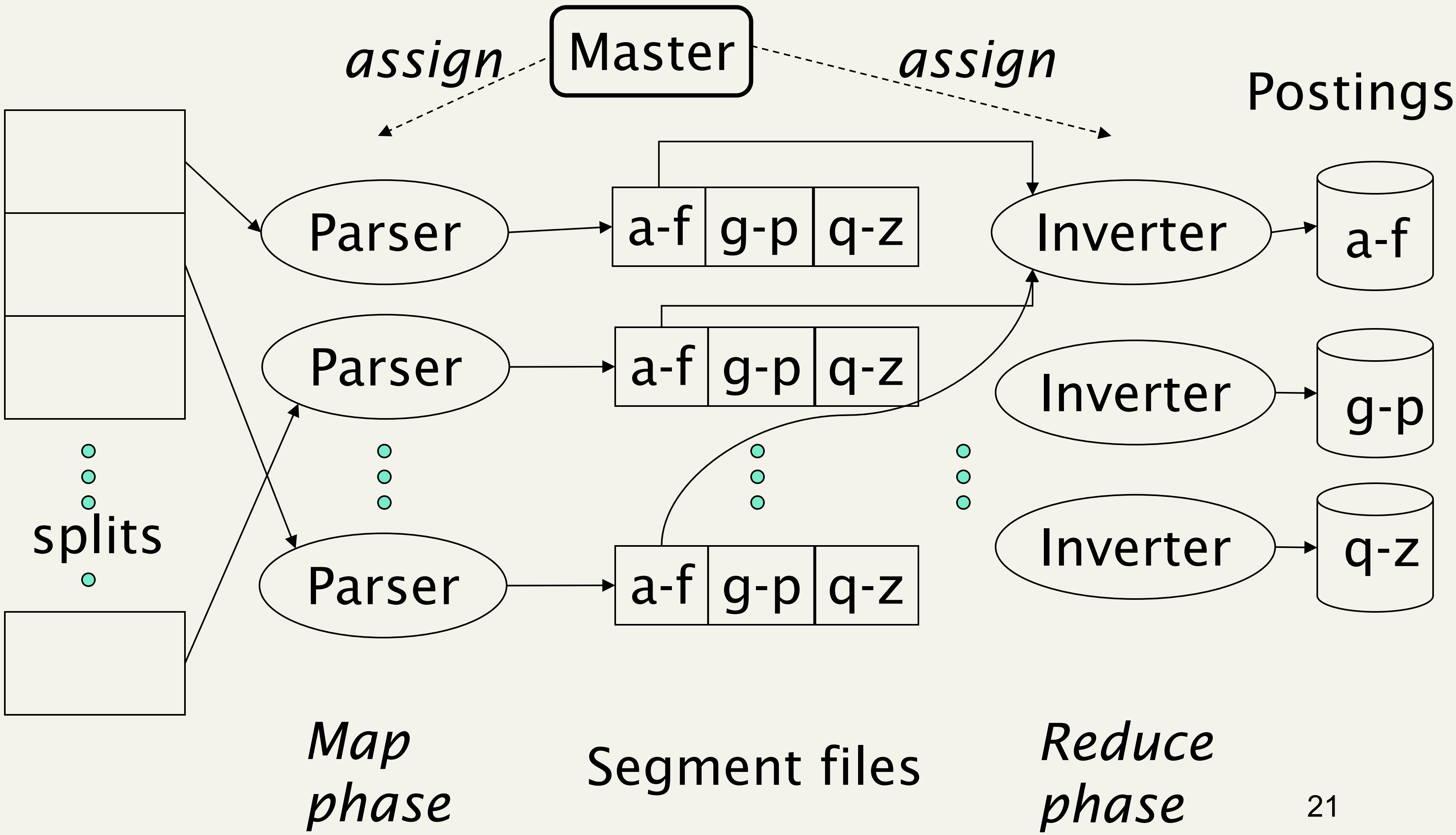
- Master assigns a split to an idle parser machine
- Parser reads a document at a time and emits (term, doc) pairs
- Parser writes pairs into *j* partitions
- Each partition is for a range of terms' first letters
  - (e.g., ***a-f, g-p, q-z***) – here *j*=3.
- Now to complete the index inversion

Inverters

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

- Parsers and inverters are not separate sets of machines.
- The same machine can be a parser (in the map phase) and an inverter (in the reduce phase).

Data flow



Master
assign
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 architecture for distributed computing …
- … without having to write code for the distribution part.

MapReduce

- MapReduce breaks a large computing problem into smaller parts by recasting it in terms of manipulation of key-value pairs
  - For indexing, (termID, docID)
- Map: mapping splits of the input data to key-value pairs
- Reduce: all values for a given key to be stored close together, so that they can be read and processed quickly
  - This is achieved by partitioning the keys into *j* terms partitions and having the parsers write key-value pairs for each term partition into a separate segment file

MapReduce

- They describe the Google indexing system (ca. 2002) as consisting of a number of phases, each implemented in MapReduce.
- Index construction was just one phase.
- Another phase: transforming a term-partitioned index into document-partitioned index.
  - *Term-partitioned:* one machine handles a subrange of terms
  - *Document-partitioned:* one machine handles a subrange of documents
- (As we discuss in the web part of the course) most search engines use a document-partitioned index … better load balancing, etc.)

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

Other sorts of indexes

- Boolean retrieval systems: docID-sorted index
  - new documents are inserted at the end of postings

- Ranked retrieval systems: impact-sorted index
  - Postings are often ordered by weight or impact
  - Postings with highest impact first
  - Insertion can occur anywhere, complicating the update of inverted index

Chapter 3

Dictionaries and Tolerant retrieval

Chapter 4

Index construction

Chapter 5

Index compression

Why compression?

- 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] is faster than [read uncompressed data]
- Premise: Decompression algorithms are fast
  - True of the decompression algorithms we use
- In most cases, retrieval system runs faster on compressed postings lists than on uncompressed postings lists.

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
- (Each postings entry is a docID)

Index parameters vs. what we index (details Table 5.1 p80)

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

Δ%: reduction in size from the previous line, except that "30 stopwords" and "150 stopwords" both use "case folding" as refereence line.
T%: cumulative (total) reduction from unfiltered

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.
- One recent research topic (Cha 7): Prune postings entries that are unlikely to turn up in the top *k* list for any query.
  - Almost no loss quality for top *k* list.

Vocabulary vs. collection size

- Can we assume an upper bound on vocabulary?
  - Not really

- Vocabulary keeps 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: $30 \le k \le 100$ and $b \approx 0.5$.
- In a log-log plot of vocabulary vs. T, Heaps' law is a line.

Heaps' Law: $M = kT^b$ Fig 5.1 p81

Vocabulary size M as a function of collection size T

For RCV1, the dashed line

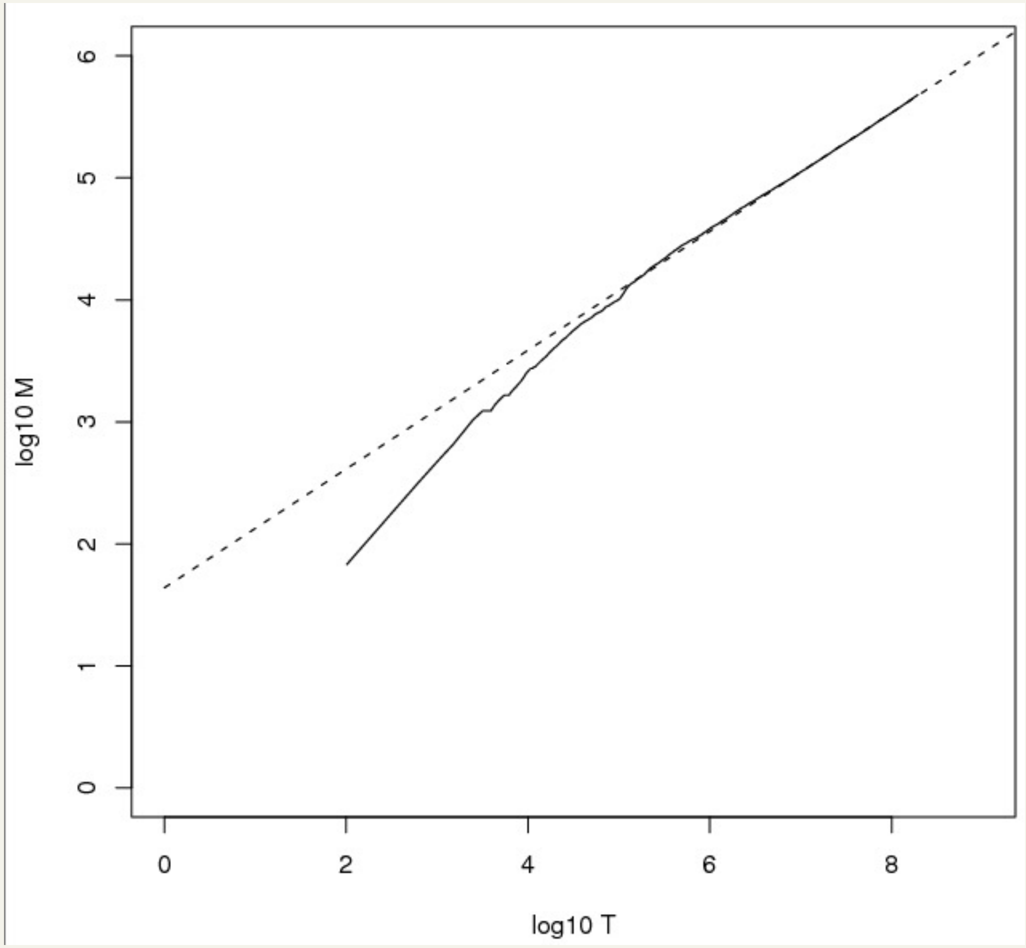
$\log_{10} M = 0.49 \log_{10} T + 1.64$

is the best least squares fit.

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

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

and b = 0.49.



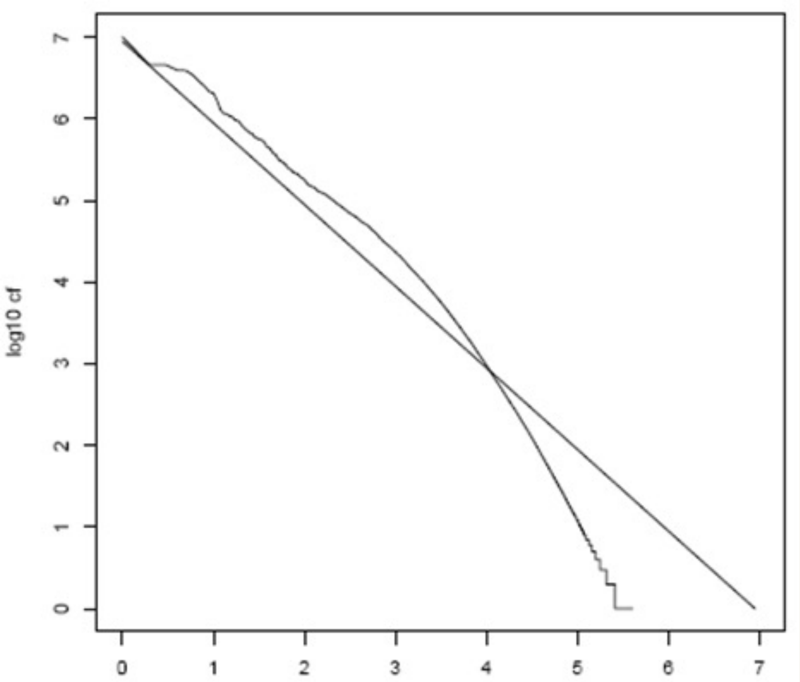
Zipf's law

- Heaps' Law gives the vocabulary size in collections.
- We also study the relative frequencies of terms.
- In a natural language, there are very few very frequent terms and very many very rare terms.
- Zipf's law: The *i*th most frequent term has frequency proportional to $1/i$.
- $\mathrm{cf}_i \propto 1/i = a/i$ where a is a normalizing constant
- cf_i is collection frequency: the number of occurrences of the term t_i in the collection.

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: $cf_i = a/i$, so
  - $\log cf_i = \log a - \log i$
  - Linear relationship

between $\log cf_i$ and $\log i$



Zipf's law for Reuters

Zipf's law: rank x frequency ~ constant

English: Rank R	Word	Frequency f	$R \times f$
10	he	877	8770
20	but	410	8200
30	be	294	8820
800	friends	10	8000
1000	family	8	8000

German: Rank R	Word	Frequency f	$R \times f$
10	sich	1,680,106	16,801,060
100	immer	197,502	19,750,200
500	Mio	36,116	18,059,500
1,000	Medien	19,041	19,041,000
5,000	Miete	3,755	19,041,000
10,000	vorläufige	1.664	16,640,000

Zipf’s law examples

Top 10 most frequent words in a large language sample:

	English			German			Spanish			Italian			Dutch	
1	the	61,847	1	der	7,377,879	1	que	32,894	1	non	25,757	1	de	4,770
2	of	29,391	2	die	7,036,092	2	de	32,116	2	di	22,868	2	en	2,709
3	and	26,817	3	und	4,813,169	3	no	29,897	3	che	22,738	3	het/'t	2,469
4	a	21,626	4	in	3,768,565	4	a	22,313	4	è	18,624	4	van	2,259
5	in	18,214	5	den	2,717,150	5	la	21,127	5	e	17,600	5	ik	1,999
6	to	16,284	6	von	2,250,642	6	el	18,112	6	la	16,404	6	te	1,935
7	it	10,875	7	zu	1,992,268	7	es	16,620	7	il	14,765	7	dat	1,875
8	is	9,982	8	das	1,983,589	8	y	15,743	8	un	14,460	8	die	1,807
9	to	9,343	9	mit	1,878,243	9	en	15,303	9	a	13,915	9	in	1,639
10	was	9,236	10	sich	1,680,106	10	lo	14,010	10	per	10,501	10	een	1,637

Dictionary compression

- Dictionary is relatively small but we want to keep it in memory
- Also, competition with other applications, cell phones, onboard computers, fast startup time
- So compression of dictionary is important
- …

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 $\log_2 800{,}000 \approx 20$ bits per docID.
- Our goal: use a lot less than 20 bits per docID.
- …

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.