AI6122 Text Data Management & Analysis

Topic: Index Construction and Compression

Index construction

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

Lucene™ Features

Lucene offers powerful features through a simple API:

Scalable, High-Performance Indexing

- over 150GB/hour on modern hardware
- small RAM requirements -- only 1MB heap
- incremental indexing as fast as batch indexing
- index size roughly 20-30% the size of text indexed

Powerful, Accurate and Efficient Search Algorithms

- ranked searching -- best results returned first
- many powerful query types: phrase queries, wildcard queries, proximity queries, range queries and more
- fielded searching (e.g. title, author, contents)
- sorting by any field
- multiple-index searching with merged results
- allows simultaneous update and searching
- flexible faceting, highlighting, joins and result grouping
- fast, memory-efficient and typo-tolerant suggesters
- pluggable ranking models, including the Vector Space Model and Okapi BM25
- configurable storage engine (codecs)

https://lucene.apache.org/

An example document collection: RCV1

- RCV1:
  - One year of Reuters newswire (part of 1995 and 1996); not very large
  - As an example for applying scalable index construction algorithms, we will use the Reuters RCV1 collection.
  - Related datasets: http://archive.ics.uci.edu/ml/datasets/Reuters+RCV1+RCV2+Multilingual,+Multiview+Text+Categorization+Test+collection#

- There are many other datasets publicly available
  - Example: English Wikipedia dump https://dumps.wikimedia.org/enwiki/
  - Example: Amazon review dataset https://nijianmo.github.io/amazon/index.html
  - Example: Yelp data challenge https://www.yelp.com/dataset/challenge

Reuters RCV1

Symbol	Statistic	Value
N	Documents	800,000
L	Average number of tokens per document	200
M	Distinct terms (word types)	400,000
	Average number of bytes per token (include spaces/punctuations)	6
	Average number of bytes per token (without spaces/punctuations)	4.5
	Average number of bytes per term	7.5
	Number of tokens	100,000,000

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.

Index Construction

- When the data collection is larger than memory can hold
  - But not so huge

- Single-pass in-memory indexing (SPIMI)
  - Key idea 1: Generate separate <u>dictionaries</u> for each block of memory
  - Key idea 2: 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 for the document collection.

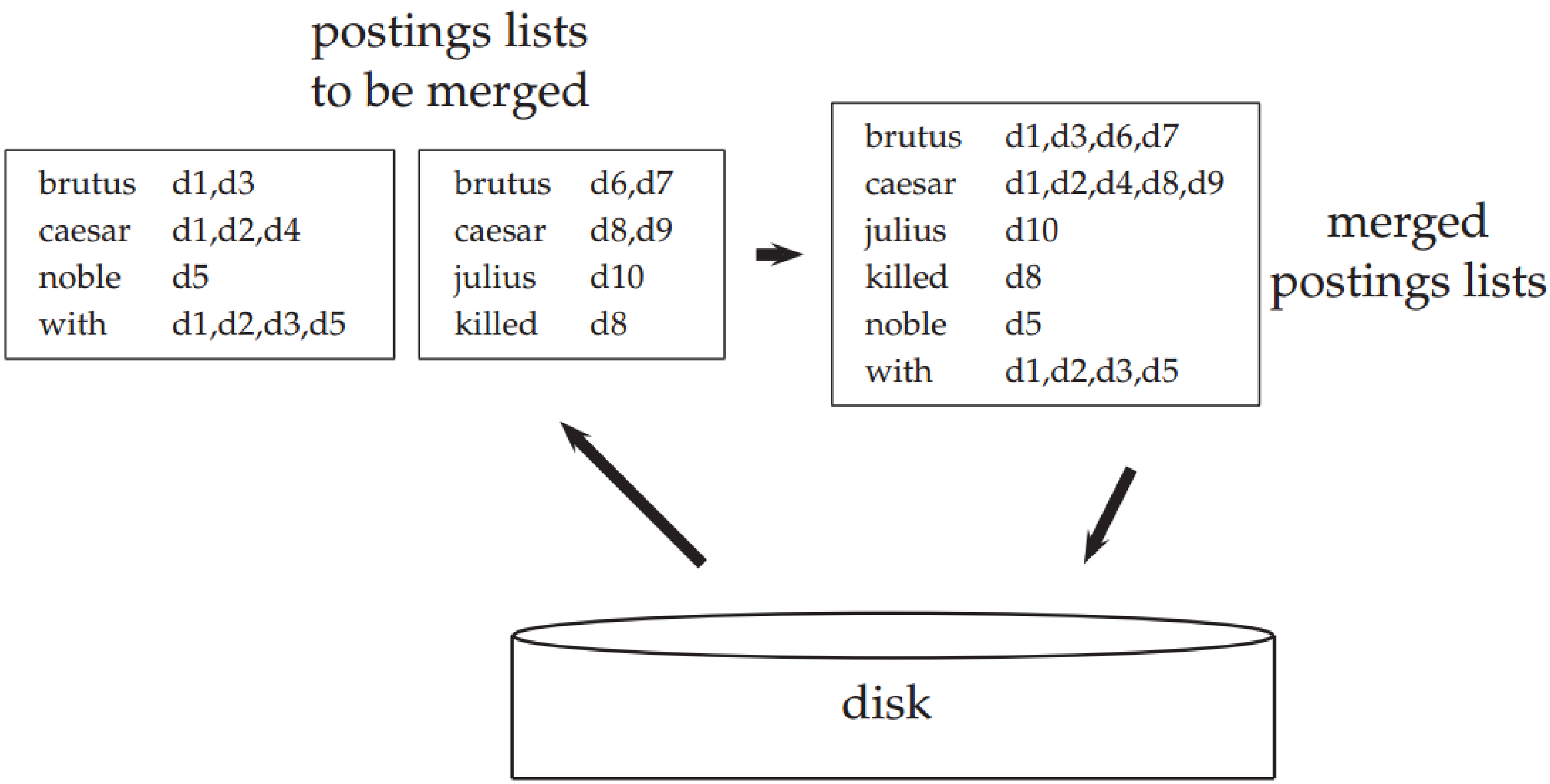
Inverted Index by SPIMI

```
SPIMI-INVERT(token_stream)
output_file = NEWFILE()
dictionary = NEWHASH()
while (free memory available)
do token ← next(token_stream)
    if term(token) ∉ dictionary
      then postings_list = ADDTODICTIONARY(dictionary, term(token))
      else postings_list = GETPOSTINGSLIST(dictionary, term(token))
    if full(postings_list)
      then postings_list = DOUBLEPOSTINGSLIST(dictionary, term(token))
    ADDTOPOSTINGSLIST(postings_list, docID(token))
sorted_terms ← SORTTERMS(dictionary)
WRITEBLOCKTODISK(sorted_terms, dictionary, output_file)
return output_file
```

Token = <term-docID> pair

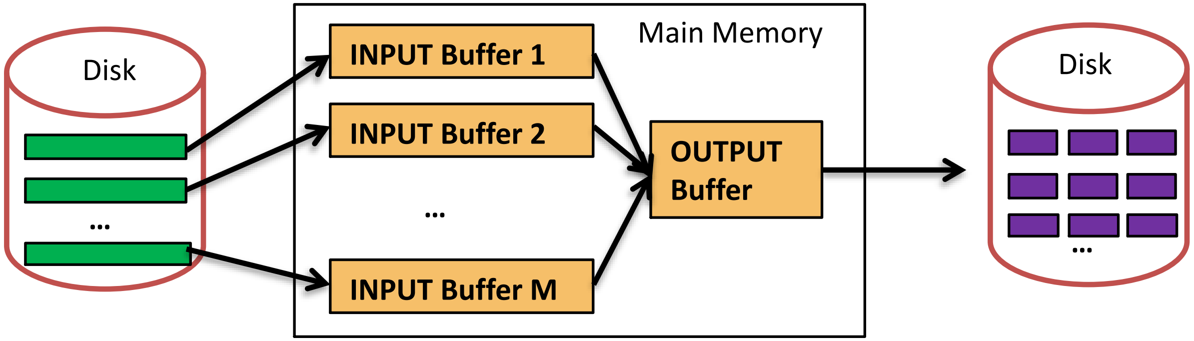
To facilitate the final merging

Merging two inverted indexes



Multi-way merge?

- Reading decent-sized chunks from all blocks simultaneously, one from each sorted block
- Merge the chunks and then write out a decent-sized output chunk



If the document collection is huge: Distributed Indexing

- Partition by documents (local index organization)
  - Documents are distributed to different subsets
  - One index is constructed for each subset of documents
  - A search query is broadcast to all indexes and results are merged
    - One machine handles a subrange of terms
  - More widely adopted in search engines

- Partition by terms (global index organization)
  - The dictionary of index terms are partitioned into subsets
  - One machine handles a subrange of terms
    - Each query term is processed by one computer node
  - Multiword queries require sending long postings between sets of nodes

- Next: how to perform term-partitioned index in parallel

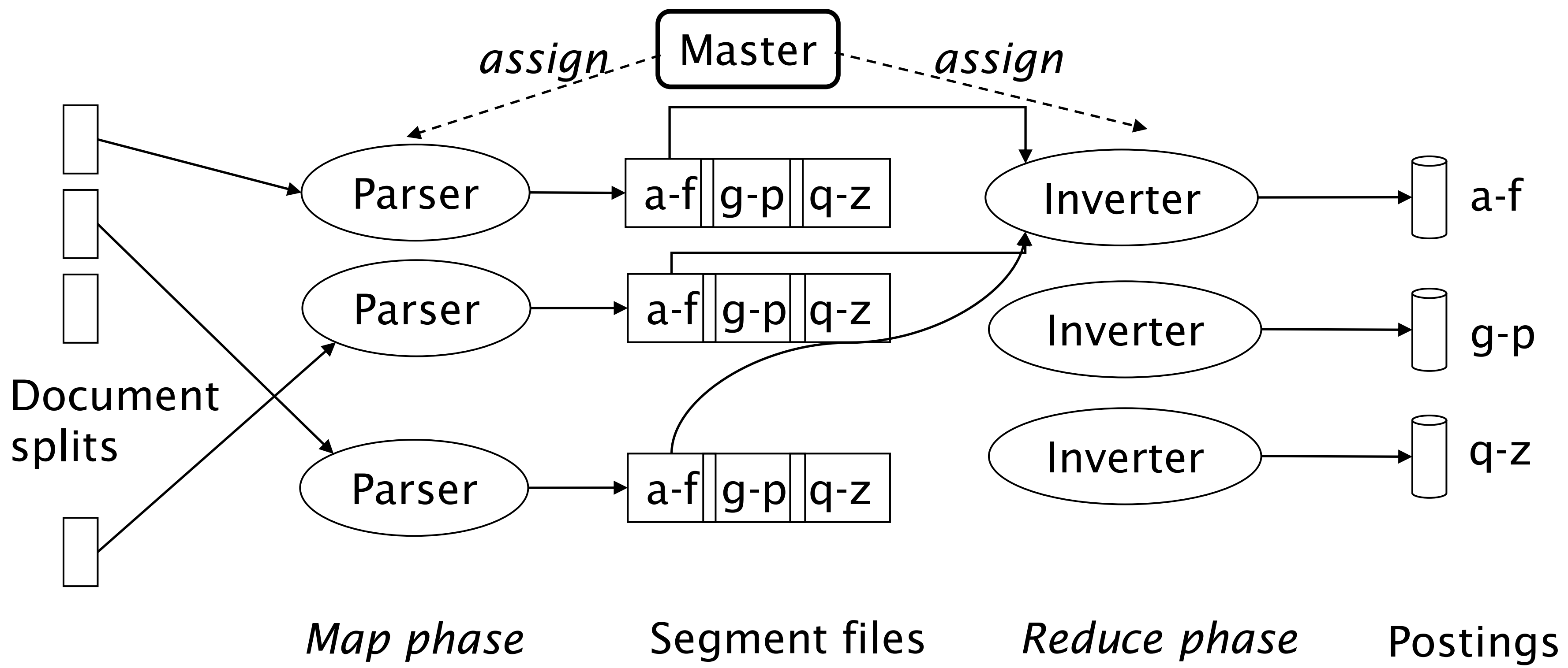
Term-partitioned distributed indexing in parallel

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

- For indexing, we use two sets of parallel tasks
  - **Parsers**
  - **Inverters**

- Break the input document collection into splits
  - Each split is a subset of documents

Term-partitioned distributed indexing: MapReduce



Parsers and Inverters

- Master assigns a split of documents to an idle parser machine

- Parser
  - reads a document at a time, and emits <term, docID> 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 $j = 3$.

- An inverter
  - collects all <term, docID> pairs for one term-partition (e.g., ***a-f***)
  - Sorts and writes to postings lists

Schema for index construction in MapReduce

- MapReduce breaks a large problem into smaller parts using
  - key-value pairs (k, v)

- Schema of map and reduce functions
  - Map phase: input → list(k, v)
  - Reduce phase: (k,list(v)) → output

- Instantiation of the schema for **index construction**
  - map: collection → list(term, docID) Parser
  - reduce: (<term1, list(docID)>, <term2, list(docID)>, …) → (postings list1, postings list2, …) Inverter

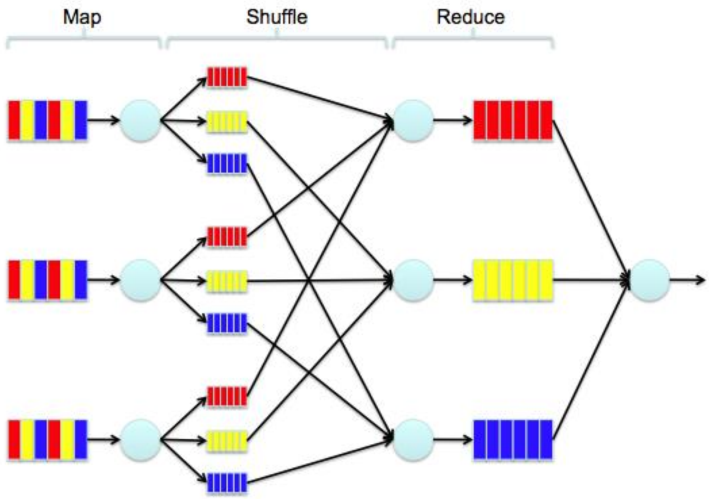
Example for index construction

- Map:
  - d1 : C came, C c’ed.
  - d2 : C died.

  → <C,d1>, <came,d1>, <C,d1>, <c’ed, d1>, <C, d2>, <died,d2>

- Reduce:
  - (<C,(d1,d2,d1)>, <died,(d2)>, <came,(d1)>, <c’ed,(d1)>)

  → (<C,(d1:2,d2:1)>, <died,(d2:1)>, <came,(d1:1)>, <c’ed,(d1:1)>)



Dynamic indexing

- Document collections may not be static
  - 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 for dynamic indexing

- Two indexes (periodically, re-index into one main index)
  - 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

- Document updates: delete and reinsert

Index Compression

- Dictionary compression
- Postings compression

BRUTUS	→	1	2	4	11	31	45	173	174	
CAESAR	→	1	2	4	5	6	16	57	132	...
CALPURNIA	→	2	31	54	101					

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, which is true of the decompression algorithms we use here

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.

  - 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.

JPG vs PNG vs EPS/PDF

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

Reuters RCV1

Symbol	Statistic	Value
N	Documents	800,000
L	Average number of tokens per document	200
M	Distinct terms (word types)	400,000
	Average number of bytes per token (include spaces/punctuations)	6
	Average number of bytes per token (without spaces/punctuations)	4.5
	Average number of bytes per term	7.5
	Number of tokens	100,000,000


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.

Index parameters vs. index size

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

Vocabulary vs. collection size

- How big is the term vocabulary?
  - That is, how many distinct words are there?
- Can we assume an upper bound?
  - All possible sequences of letters of length 20?
- In practice, the vocabulary will keep growing with the collection size
  - Especially with Unicode ☺

Vocabulary vs. collection size

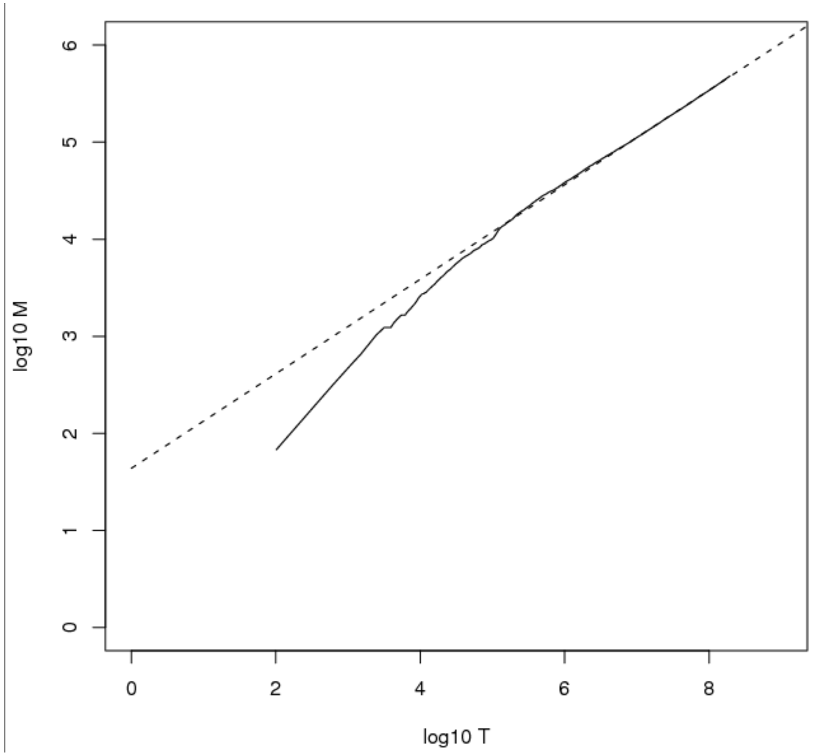
- How big is the term vocabulary (distinct words)?

- 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: $M = kT^b$

- For RCV1, the dashed line is the best least squares fit.
  - $\log_{10} M = 0.49 \log_{10} T + 1.64$
  - $M = 10^{1.64} T^{0.49}$
  - $k = 10^{1.64} \approx 44$ and b = 0.49.
  - Good empirical fit for Reuters RCV1 !



- Example:
  - for first 1,000,020 tokens, law predicts 38,323 terms;
  - Actual number: 38,365 terms

Zipf's law

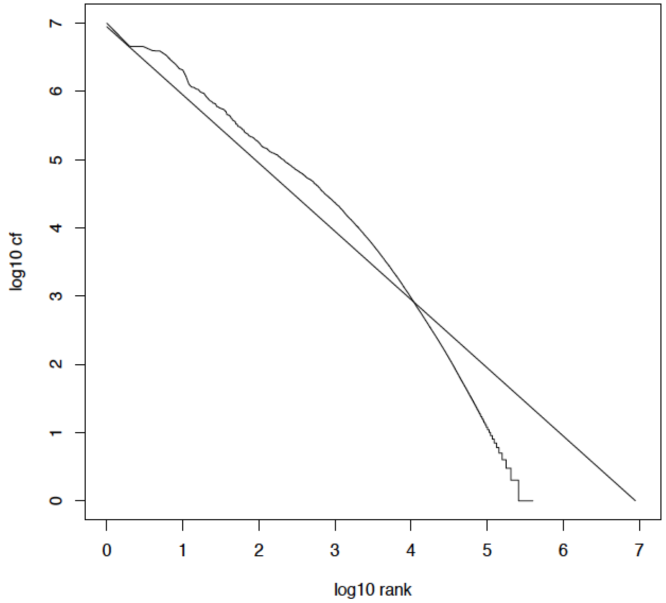
- Heaps' law gives the vocabulary size in collections.

- 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.

Zipf consequences

- If the most frequent term (the) occurs $cf1$ times
  - then the second most frequent term (of) occurs $cf1/2$ times
  - the third most frequent term (and) occurs $cf1/3$ 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



Index Compression

- Now, we will consider compressing the space for the dictionary and postings
  - Basic Boolean index only
  - Not considering positional indexes, etc.
  - We will consider different compression schemes

- 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

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.

….systilesyzygeticsyzygialsyzygyszaibelyiteszczecinszomo….

DocFreq.	Postings ptr.	Term ptr.
33	→	→
29	→	→
44	→	→
126	→	→

Total string length =
400K x 8B = 3.2MB

Pointers resolve 3.2M positions:
$\log_2 3.2\text{M}$ = 22bits = 3 Bytes

Space for dictionary as a string

- Storage requirement:
  - 4 bytes per term for document frequency
  - 4 bytes per term for pointer to Postings.
  - 3 bytes per term pointer
  - Avg. 8 bytes per term in term string

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

- Total: 400K terms x 19 $\Rightarrow$ 7.6 MB

Can we do better? → Blocking

- Store pointers to every kth term string.
  - Example below: k=4.

- Need to store term lengths (1 extra byte)

….**7*systile*9*syzygetic*8*syzygial*6*syzygy*11*szaibelyite*8*szczecin*9*szomo***….

Lose 4 bytes per 4 terms

Save 9 bytes per 4 terms

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

Is Blocking effective

- Example for block size k = 4
  - Without blocking, we used 3 bytes/pointer: 3 x 4 = 12 bytes for every 4 terms
  - With blocking, we use 3 + 4 = 7 bytes for every 4 terms
  - This reduces the size of the dictionary from 7.6 MB to 7.1 MB.

- Shall we use larger k?
  - Better compression
  - Slower term lookup

Front coding – more compression

- Sorted words commonly have long common prefix
  - Store differences only, for last $k-1$ in a block of k
  - 8automata8automate9automatic10automation

→8***automat****a1◊***e***2◊***ic***3◊***ion***

Encodes ***automat***

Extra length beyond ***automat.***

- For RCV1 dictionary compression
  - Dictionary-as-String with pointers to every term, 7.6M
  - with blocking k = 4, 7.1M
  - With Blocking + front coding 5.9M

Postings compression

- The postings file is much larger than the dictionary
  - Factor of at least 10.
  - Compression: store each posting compactly.

- A posting for our purposes is a docID.
  - For Reuters (800,000 documents), we use 32 bits per docID when using 4-byte integers.
  - Alternatively, we can use log2 800,000 ≈ 20 bits per docID.

- Our goal: use far fewer 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 $\log_2 1M \sim 20$ bits.

- A term like **<u>the</u>** occurs in virtually every doc, so 20 bits per 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.

	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 ~log2G 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 code example (we skip details)

1100111000 101 110100011000110001

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.

RCV1 Index Compression

Data	Size in MB
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
collection (text)	960
postings, uncompressed (32-bit words)	400
postings, uncompressed (20 bits)	250
postings, variable byte encoded	116
postings, γ–encoded (a coding scheme seldom used in practice)	101

Index compression summary

- We can now create an index for highly efficient Boolean retrieval that is very space efficient
  - However, we've ignored positional information

- Hence, space savings are less for indexes used in practice
  - But techniques substantially the same.