Web Information Retrieval

Lecture 4

Dictionaries, Index Compression

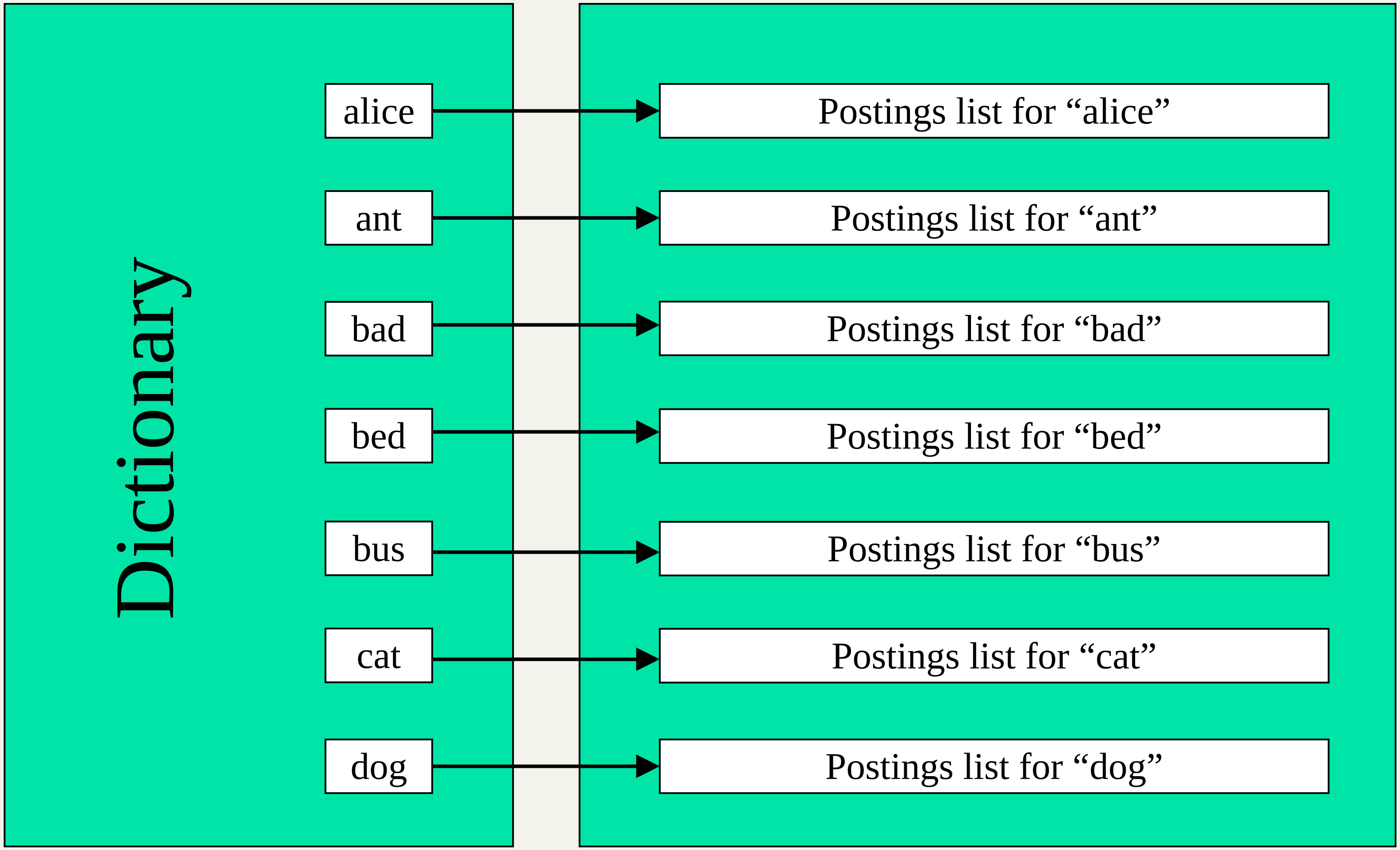
Recap: lecture 2,3

- Stemming, tokenization etc.
- Faster postings merges
- Phrase queries
- Index construction

This lecture

- Dictionary data structure
- Index compression

Entire data structure



A naïve dictionary

- An array of records:

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

char[20] int Postings *

20 bytes 4/8 bytes 4/8 bytes

- How do we quickly look up elements at query time?

Exercises

- Is binary search really a good idea?
- What are the alternatives?

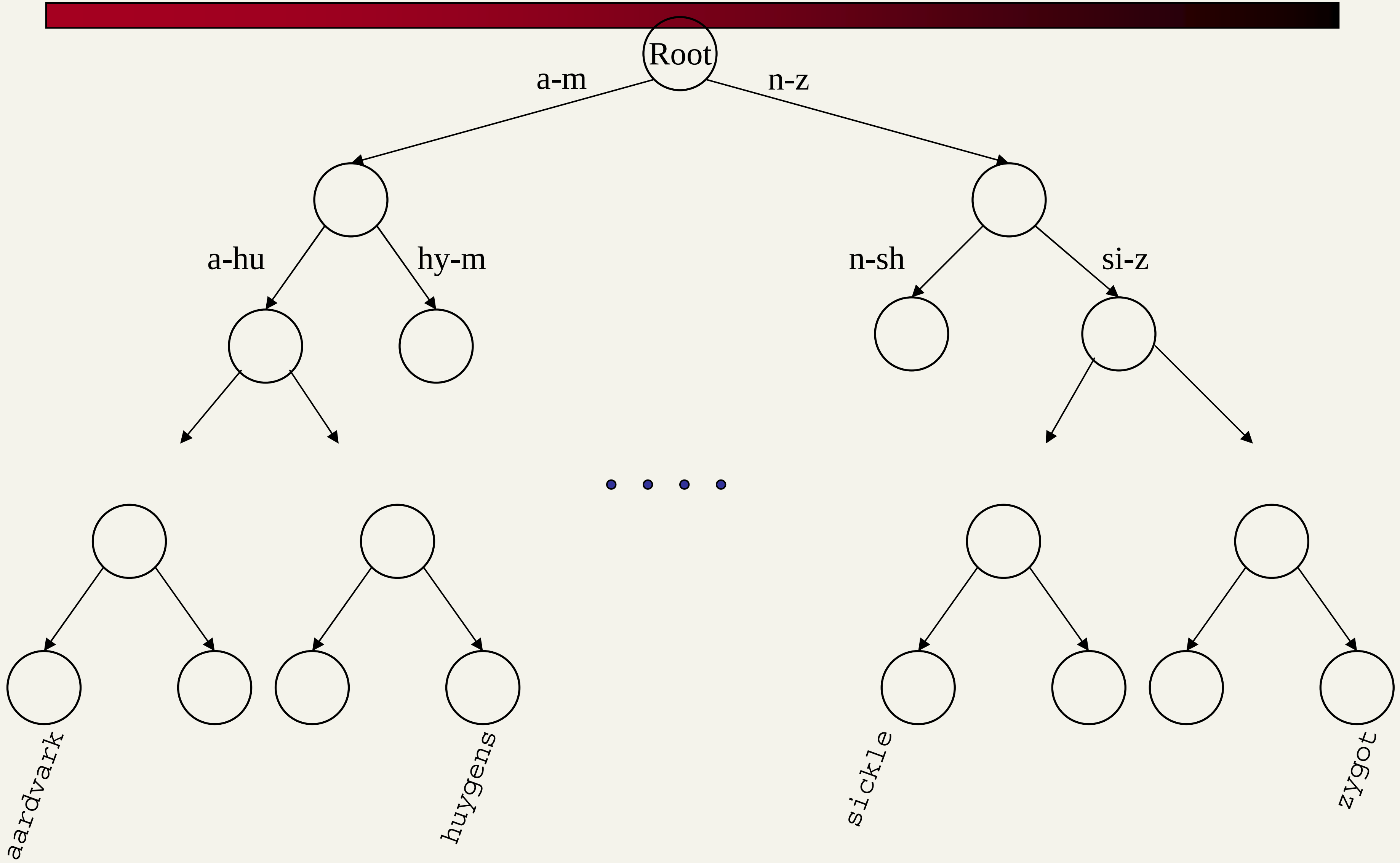
Dictionary data structures

- Two main choices:
  - Hashtables
  - Trees
- Some IR systems use hashtables, some trees

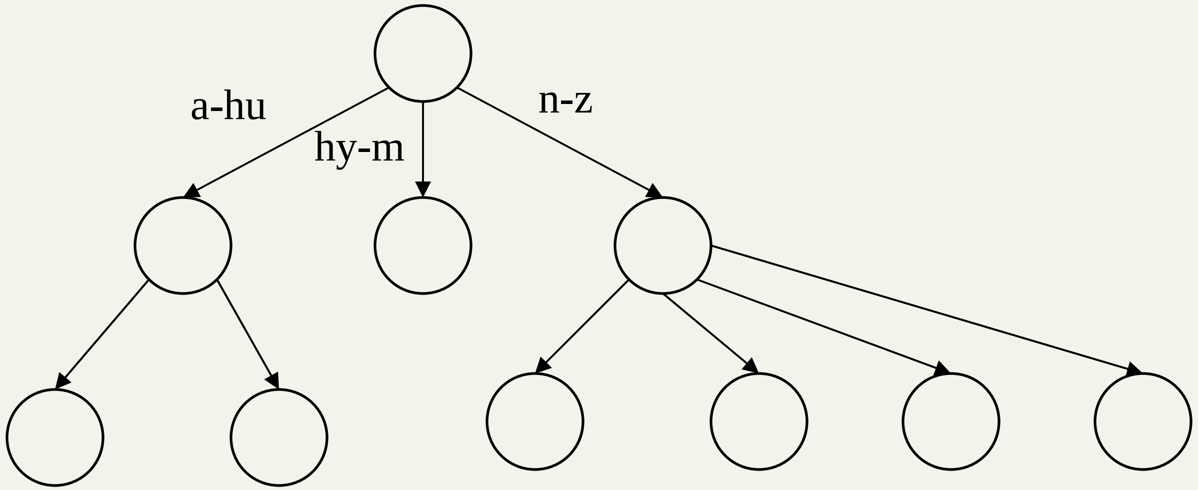
Hashtables

- 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 [tolerant retrieval]
  - If vocabulary keeps growing, need to occasionally do the expensive operation of rehashing *everything*

Tree: binary tree



Tree: 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].

Trees

- Simplest: binary tree
- More usual: B-trees
- Trees require a standard ordering of characters and hence strings … but we typically have one
- Pros:
  - Solves the prefix problem (terms starting with *hyp*)
- Cons:
  - Slower: O(log *M*) [and this requires *balanced* tree]
  - Rebalancing binary trees is expensive
    - But B-trees mitigate the rebalancing problem

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

Compression: Two alternatives

- Lossless compression: all information is preserved, but we try to encode it compactly
  - What IR people mostly do
- Lossy compression: discard some information
  - Using a stopword list can be viewed this way
  - Techniques such as Latent Semantic Indexing (later) can be viewed as lossy compression
  - One could prune from postings entries unlikely to turn up in the **top *k*** list for query on word
    - Especially applicable to web search with huge numbers of documents but short queries (e.g., Carmel et al. *SIGIR 2002)*

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
T	non-positional postings	100,000,000

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

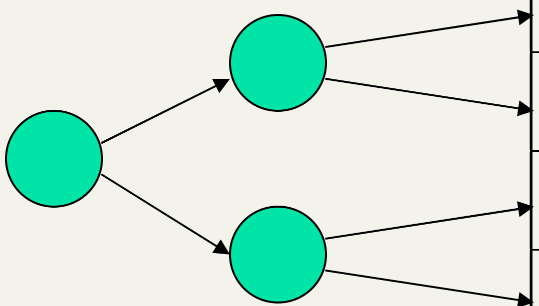
DICTIONARY COMPRESSION

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.



Terms	**Freq.**	**Postings ptr.**
a	656,265	
aachen	65	
….	….	
zulu	221	

Dictionary search structure

20 bytes

4 bytes each

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.*
- Written English averages ~4.5 characters/word.
  - Exercise: Why is/isn't this the number to use for estimating the dictionary size?
- Ave. dictionary word in English: ~8 characters
  - How do we use ~8 characters per dictionary term?
- Short words dominate token counts but not type average.

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….

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

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

Pointers resolve 3.2M positions: $\log_2 3.2M$ = 22bits = 3bytes

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)

Now avg. 11 bytes/term, not 20.

Blocking

- Store pointers to every *k*th 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***….

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

Save 9 bytes on 3 pointers.

Lose 4 bytes on term lengths.

Front coding

- Front-coding:
  - Sorted words commonly have long common prefix – store differences only
  - (for last *k-1* in a block of *k*)

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

→8***automat*a***1◊***e***2◊***ic***3◊***ion***

Encodes ***automat***

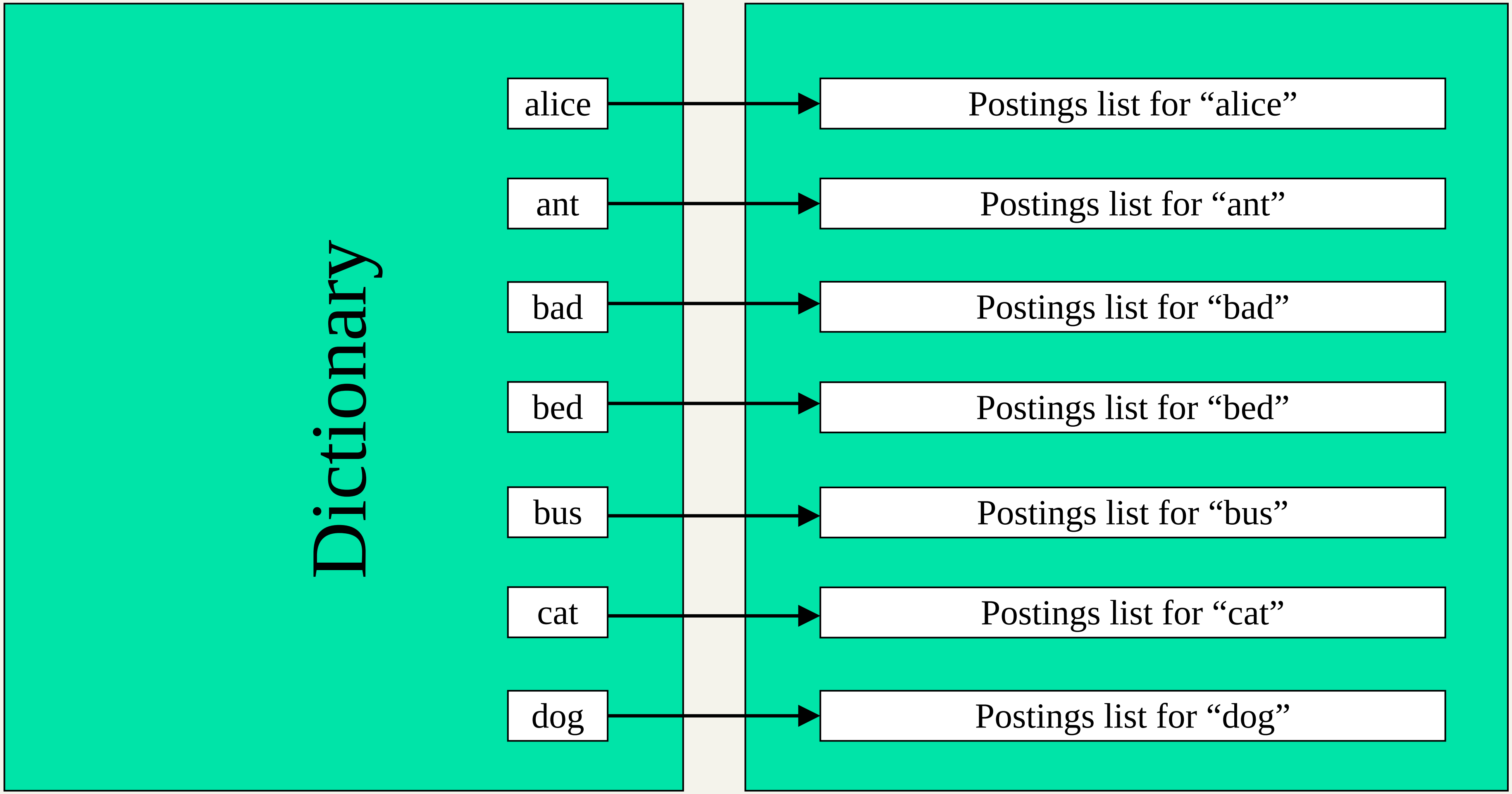
Extra length beyond ***automat.***

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

Entire data structure



Details (no compression)

Term	Freq.	Postings ptr.
alice	56,265	
…	…	
ant	658,452	
…	…	

3 | 19 | 25 | 33 | 48 | 57 | 70 | 71 | 89 | …

6 | 10 | 22 | 40 | 46 | 66 | 69 | 87 | 94 | …

Postings list for “bad”

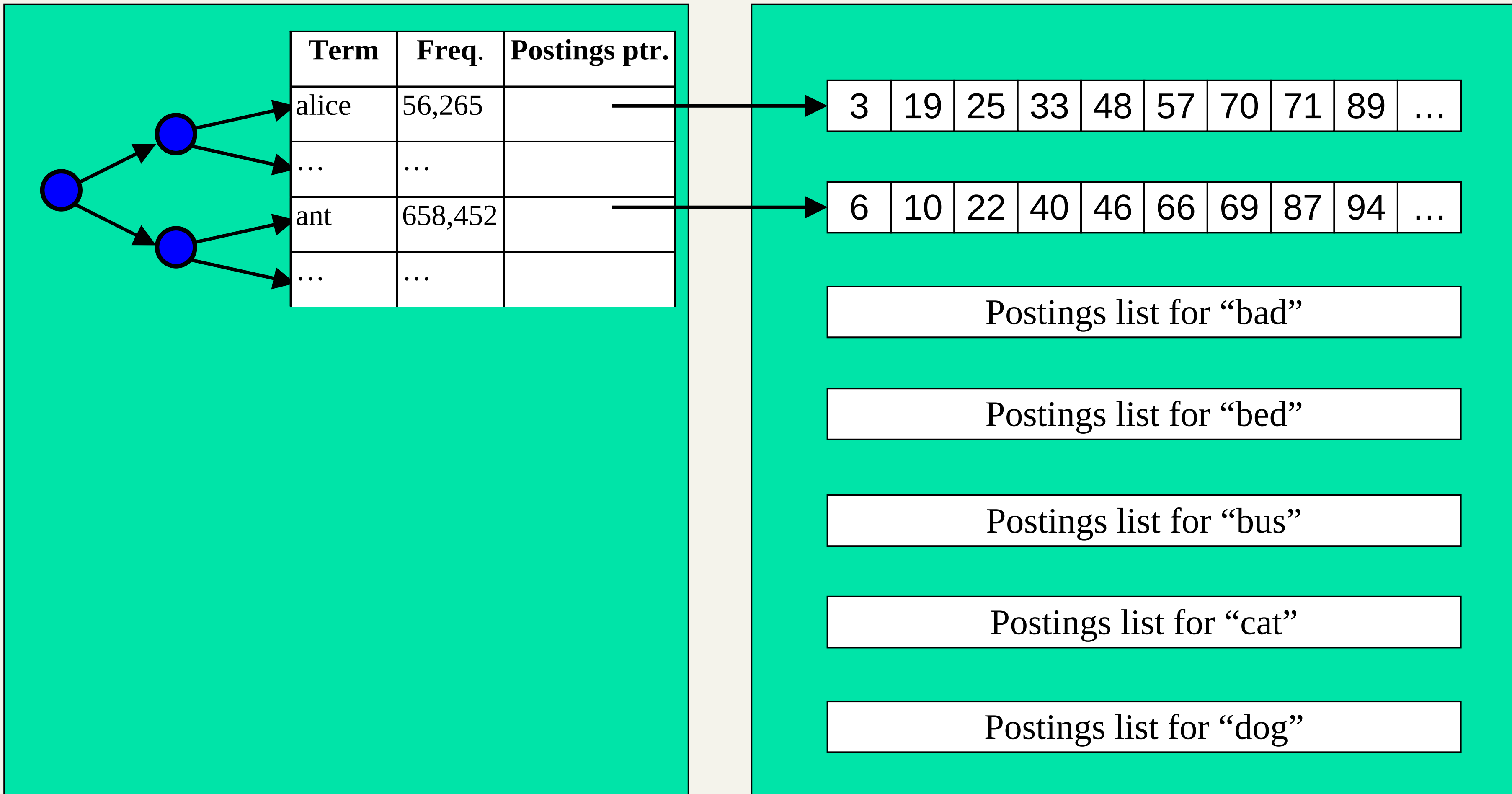
Postings list for “bed”

Postings list for “bus”

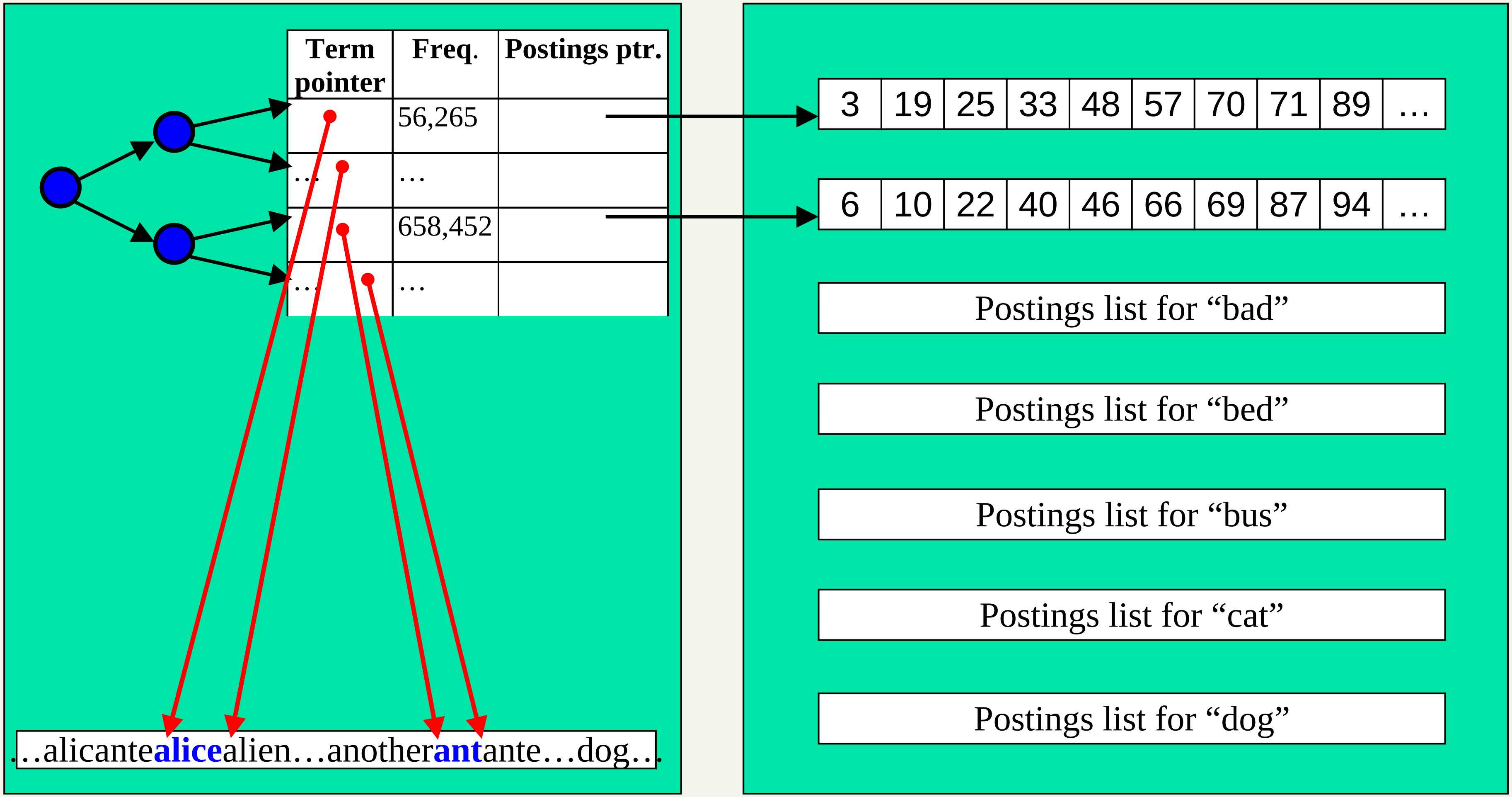
Postings list for “cat”

Postings list for “dog”

Details (no compression)



Details (dictionary compression)



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 $\log_2 800{,}000 \approx 20$ bits per docID.
- Our goal: use far fewer than 20 bits per docID.

Storage analysis

- First will consider space for postings pointers
- Basic Boolean index only
  - Devise compression schemes
- Then will do the same for dictionary
- No analysis for positional indexes, etc.

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

- Store list of docs containing a term in increasing order of doc id.
  - ***Brutus***: 33,47,154,159,202 …
- Consequence: suffices to store *gaps*.
  - 33,14,107,5,43 …
- Hope: most gaps encoded with far fewer than 20 bits.

Postings file entry

- Store list of docs containing a term in increasing order of doc id.
  - ***Brutus***: 33,47,154,159,202 …
- Consequence: suffices to store *gaps*.
  - 33,14,107,5,43 …
- Hope: most gaps encoded with far fewer than 20 bits.

Postings file entry

- Store list of docs containing a term in increasing order of doc id.
  - ***Brutus***: 33,47,154,159,202 …
- Consequence: suffices to store *gaps*.
  - 33,14,107,5,43 …
- Hope: most gaps encoded with far fewer than 20 bits.

Variable encoding

- For ***arachnocentric***, will use ~20 bits/gap entry.
- For ***the***, will use ~1 bit/gap entry.
- If the average gap for a term is G, want to use ~$\log_2 G$ bits/gap entry.
- <u>Key challenge</u>: encode every integer (gap) with ~ as few bits as needed for that integer.

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

Encoding types

There are 2 types of encodings:

- Variable byte encodings

  Minimize number of bytes used
- Bit-level encodings

  Minimize number of bits used

Encoding types

There are 2 types of encodings:

- **Variable byte encodings**

  Minimize number of bytes used
- Bit-level endodings

  Minimize number of bits used

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 <u>continuation</u> 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
  - Good low-tech blend of variable-length coding and sensitivity to computer memory alignment matches (vs. bit-level codes, which we look at next).

Encoding types

There are 2 types of encodings:

- Variable byte encodings
  Minimize number of bytes used
- Bit-level encodings
  Minimize number of bits used

Encoding types

There are 2 types of encodings:

- Variable byte encodings
  Minimize number of bytes used
- **Bit-level encodings**
  Minimize number of bits used

γ (gamma) codes for gap encoding

Length	Offset

- Represent a gap G as the pair <*length*,*offset*>
- *length* is in unary and uses $\lfloor \log_2 G \rfloor + 1$ bits to specify the length of the binary encoding of
- *offset* = $G - 2^{\lfloor \log_2 G \rfloor}$ in binary.

Recall that the unary encoding of x is a sequence of x 1's followed by a 0.

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 80 is:

1111111111111111111111111111111111111111111
111111111111111111111111111111111111110

- This doesn’t look promising, but….

γ codes

- We can compress better with bit-level codes
  - The γ 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.
- γ code of 13 is the concatenation of *length* and *offset*: 1110101

γ codes for gap encoding

- e.g., 9 represented as <1110,001>.
- 2 is represented as <10,0>.
- <u>Exercise</u>: does zero have a γ code?

Exercise

- Given the following sequence of γ –coded gaps, reconstruct the postings sequence:

1110001110101011111101101111011

From these γ–decode and reconstruct gaps, then full postings.

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

γ 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

What we've just done

- Encoded each gap as tightly as possible, to within a factor of 2.
- For better tuning (and a simple analysis) - need a handle on the <u>distribution</u> of gap values.

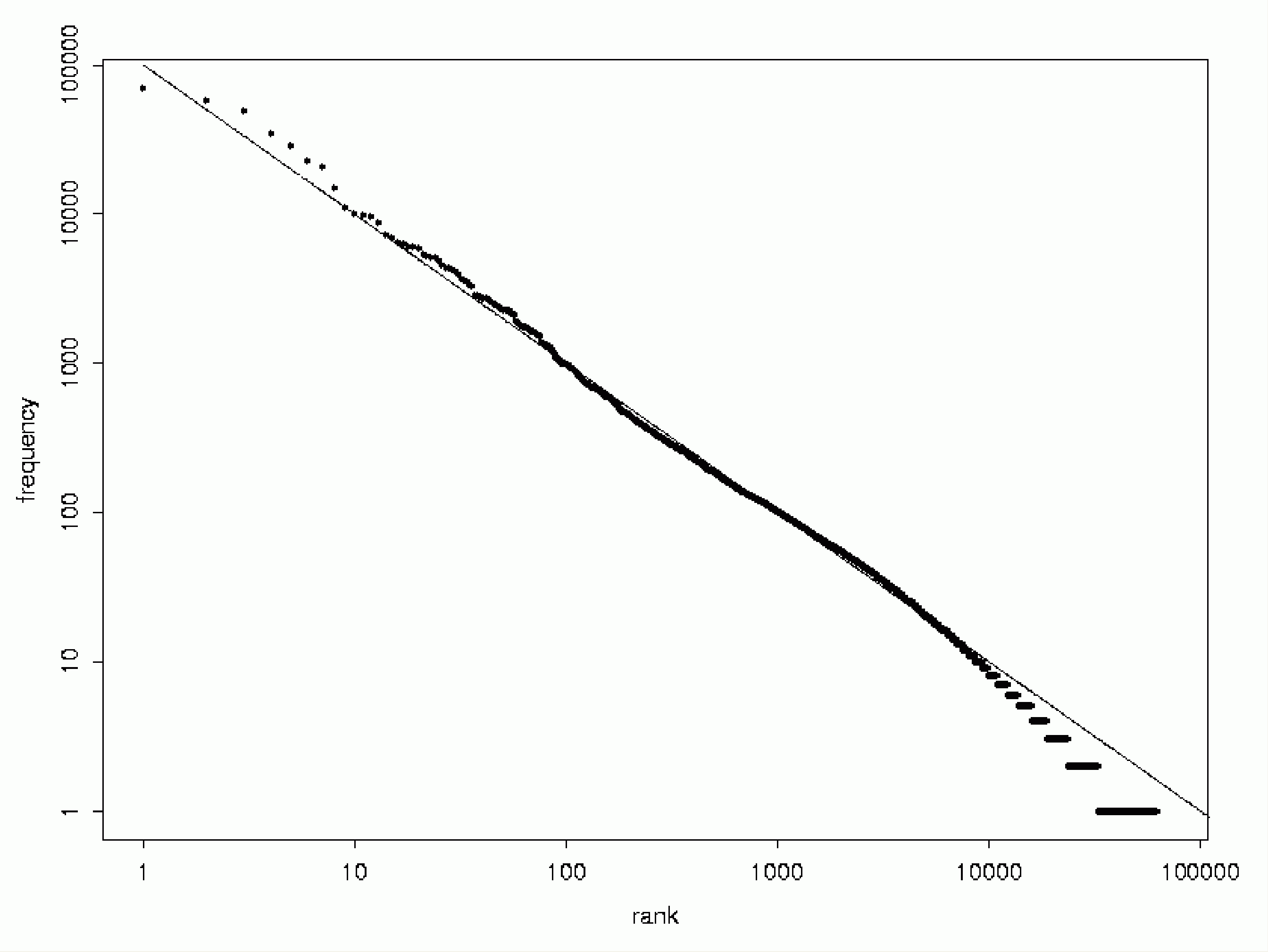
Analysis

- To analyze the space used we need to know the distribution of the word frequencies
- This approximately follows Zipf’s law

Zipf's law

- The *i*-th most frequent term has frequency proportional to *1/i*
- Use this for a crude analysis of the space used by our postings file pointers
  - Not yet ready for analysis of dictionary space

Zipf’s law log-log plot



Rough analysis based on Zipf

- The *i*-th most frequent term has relative frequency proportional to $\frac{1}{i}$
- Let this relative frequency be $\frac{c}{i}$
- Then $\sum_{i=1}^{M(=400K)} \frac{c}{i} = 1$
- The *M*-th **Harmonic number** is $H_M = \sum_{i=1}^{M} \frac{1}{i} \approx \ln M$
- Thus $c = \frac{1}{H_M}$ which is $\approx \frac{1}{\ln M} = \frac{1}{\ln 400K} \approx \frac{1}{13}$
- So the *i*-th most frequent term has frequency roughly $\frac{c}{i} \approx \frac{1}{13i}$

Postings analysis contd.

- Expected number of occurrences of the *i* th most frequent term in a doc of length L = 200 is:

$$L\frac{c}{i} \approx L\frac{13}{i} = 200\frac{13}{i} \approx \frac{15}{i}$$

- Let $Q = Lc \approx 15$
- Then the Q most frequent terms are likely to occur in every document.
- The second Q most frequent terms are likely to occur in every 2 documents.
- Now imagine the term-document incidence matrix with rows sorted in decreasing order of term frequency:

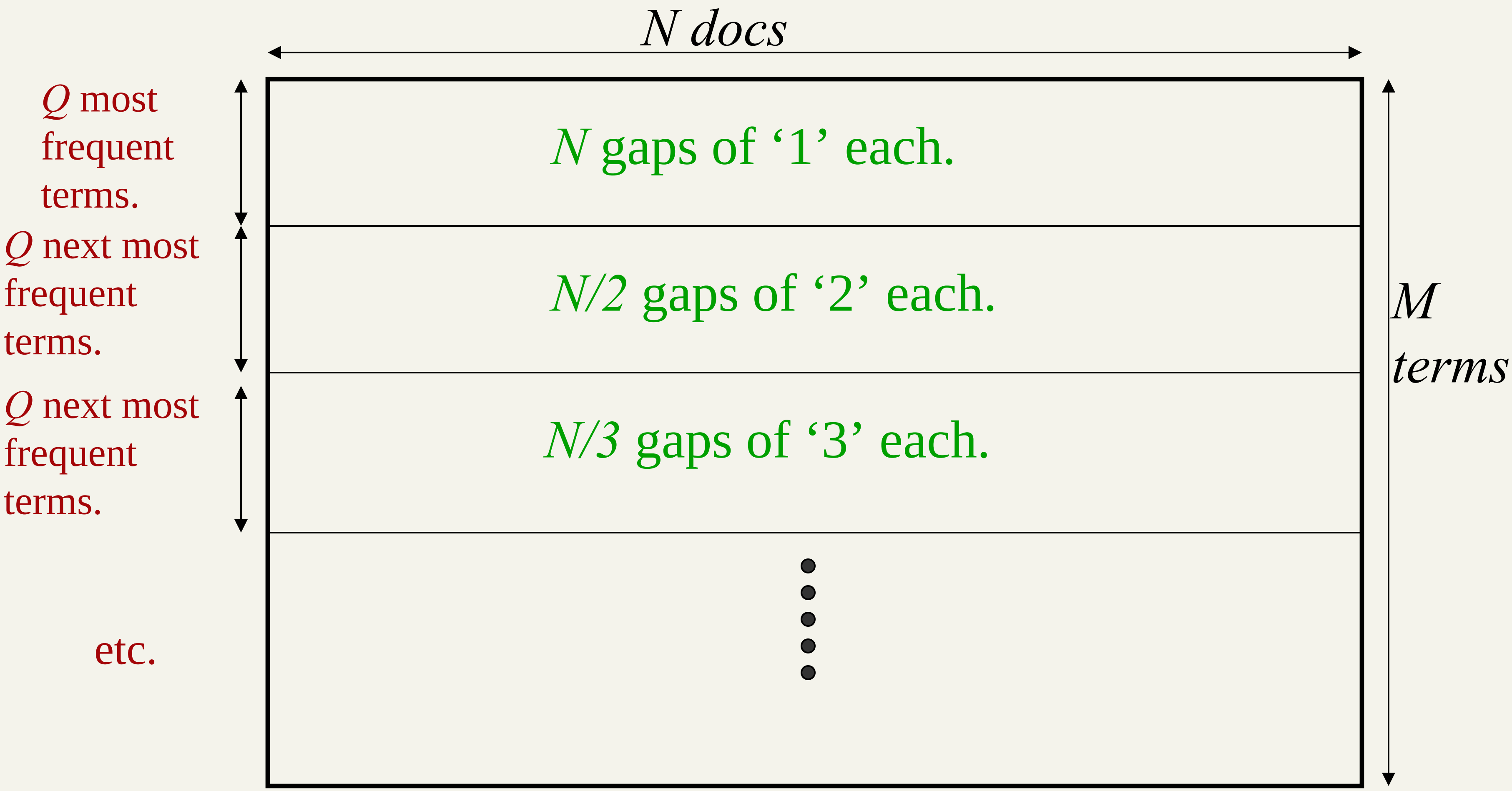
Postings analysis contd.

- Expected number of occurrences of the *i* th most frequent term in a doc of length *L* = 200 is:

$$L\frac{c}{i} \approx L\frac{.1}{i} = 200\frac{.1}{i} \approx \frac{15}{i}$$

- Let $Q = Lc \approx 15$
- Then the *Q* most frequent terms are likely to occur in every document.
- The second *Q* most frequent terms are likely to occur in every 2 documents.
- Now imagine the term-document incidence matrix with rows sorted in decreasing order of term frequency:

Rows by decreasing frequency



Q-row blocks

- In the j-th of these Q-row blocks, we have Q rows each with Q/i gaps of i each.
- Encoding a gap of i takes us $\log_2 i + 1 \approx \log_2 i$ bits
- So such a row uses space $\approx \dfrac{2N \log_2 i}{i}$ bits.
- For the entire block:

$$\approx \frac{2NQ \log_2 i}{i} = \frac{2 \cdot 800,000 \cdot 15 \log_2 i}{i} = \frac{2.4 \cdot 10^7 \log_2 i}{i} \text{ bits}$$

- Total:

$$\approx \sum_{j=1}^{M/Q} \frac{2.4 \cdot 10^7 \log_2 i}{i} = \sum_{j=1}^{400{,}000/15} \frac{2.4 \cdot 10^7 \log_2 i}{i} = 1.8 \cdot 10^9 \text{ bits} = 225\text{GB}$$

Exercise

- So we've taken 1GB of text and produced from it a 225MB index that can handle Boolean queries!
- It is an approximation. In practice, if we try γ encoding for RCV1 we compress it to 101MB

Make sure you understand <u>all</u> the approximations in our probabilistic calculation.

Caveats

- Assumes Zipf's law applies to occurrence of terms in docs.
- All gaps for a term taken to be the same.
- Does not talk about query processing.
- This is not the entire space for our index:
  - does not account for dictionary storage
  - as we get further, we'll store even more stuff in the index

Exercise

- How would you adapt the space analysis for γ – coded indexes to the scheme using continuation bits?

Exercise (harder)

- How would you adapt the analysis for the case of positional indexes?
- Intermediate step: forget compression. Adapt the analysis to estimate the number of positional postings entries.

γ 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

Resources

- IIR Chapters 3.1, 5