INDEX

Chapter 5

Inverted File

- An inverted file is a list of search terms that are organized for associative look-up to answer the questions:
  - In which document does a specified search term appear ?
  - Where within each document does each term appear?
- Usually, this information is captured using the ***word list*** and the ***posting file*** which together form the inverted file system.
- These files are used to store calculated weights and information about the position of terms in the document.

Example – Simple inverted index

and	1
aquarium	3
are	3 4
around	1
as	2
both	1
bright	3
coloration	3 4
derives	4
due	3
environments	1
fish	1 2 3 4
fishkeepers	2
found	1
fresh	2
freshwater	1 4
from	4
generally	4
in	1 4
include	1
including	1
iridescence	4
marine	2
often	2 3
only	2
pigmented	4
popular	3
refer	2
referred	2
requiring	2
salt	1 4
saltwater	2
species	1
term	2
the	1 2
their	3
this	4
those	2
to	2 3
tropical	1 2 3
typically	4
use	2
water	1 2 4
while	4
with	2
world	1

Given Q = “Tropical fish”

S1= 1,2,3,4

S2= 1,2,3

S1 ∩ S2 = 1,2,3

S_1 Tropical fish include fish found in tropical environments around the world, including both freshwater and salt water species.

S_2 Fishkeepers often use the term tropical fish to refer only those requiring fresh water, with saltwater tropical fish referred to as marine fish.

S_3 Tropical fish are popular aquarium fish, due to their often bright coloration.

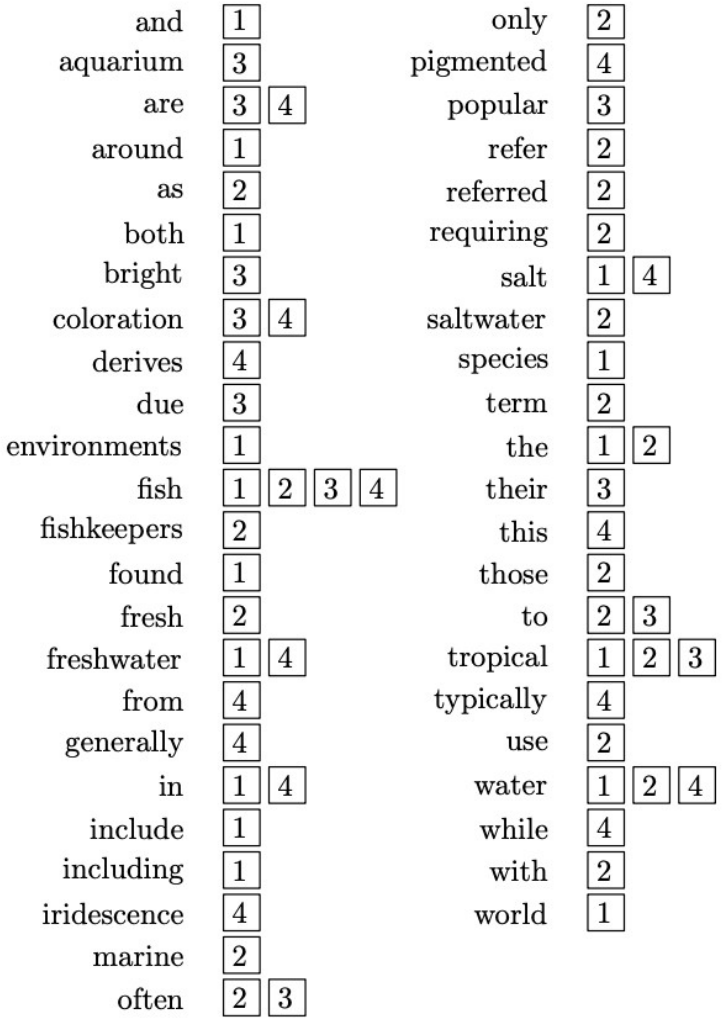
S_4 In freshwater fish, this coloration typically derives from iridescence, while salt water fish are generally pigmented.

- For each term, the lists the documents that contain that term (with no other information)

- No frequency information is contained.
- So while “fish” appears in S_2 twice but in S_1 only once, the index does not differentiate the two documents.

Lexicographic Order

- It is important that the word list can be processed sequentially, i.e, in alphabetic order.
- To search with wild cards, e.g. *comp*,* which expands to every term beginning with the letters "comp".
- To list results for browsing lists of search terms.



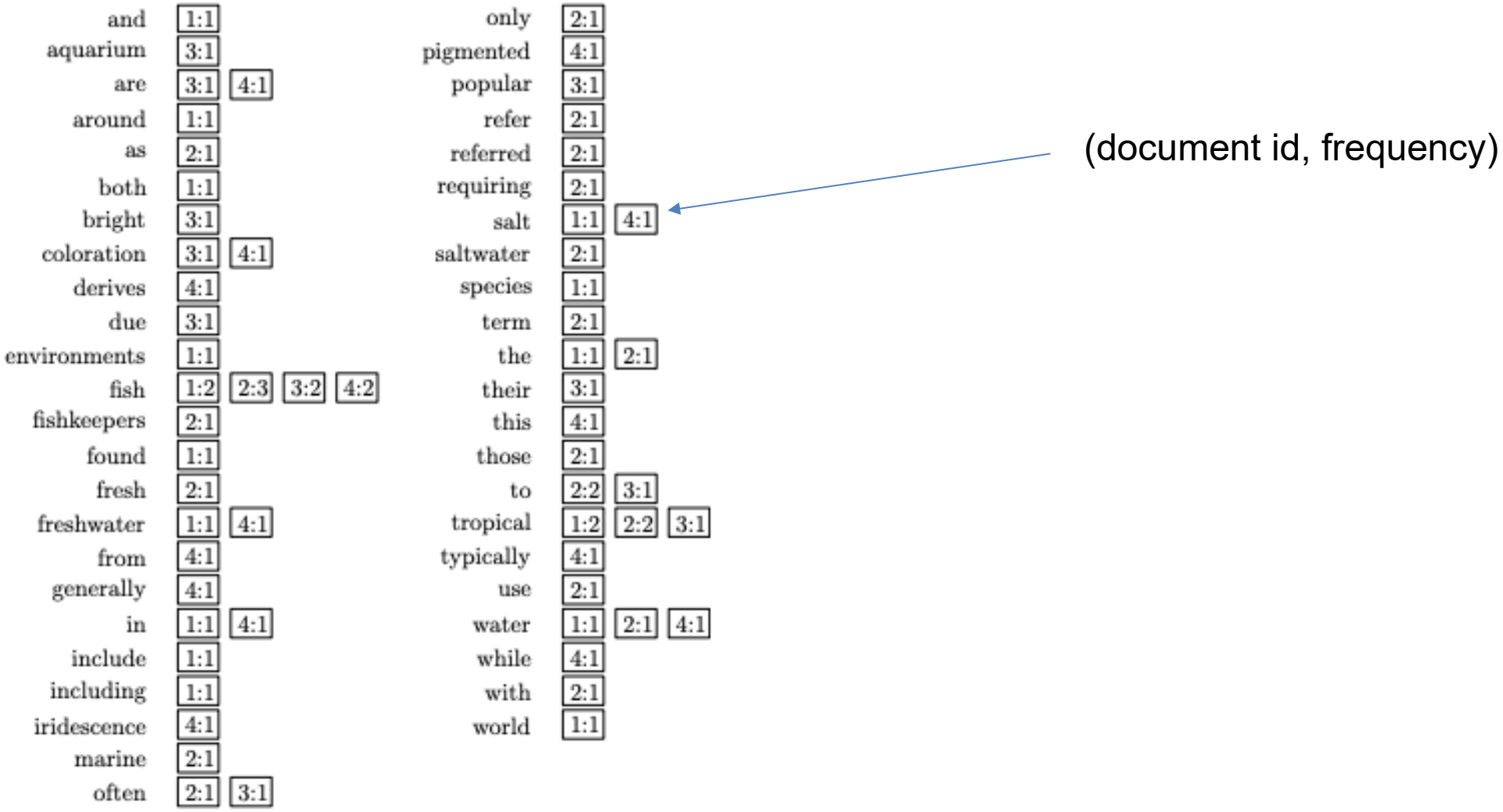
Use of Inverted Files for Calculating Similarities

- In the term vector space, if **q** is query and $\mathbf{d}_j$ a document, then **q** and $\mathbf{d}_j$ have no terms in common iff $\mathbf{q}.\mathbf{d}_j = 0$.
-
  - 1. To calculate all the **non-zero similarities** find *R.* the set of all the documents, $\mathbf{d}_j$, that contain **at least one term** in the query:
  - 2. Merge the inverted lists for each term t_i in the query, with a logical *or*, to establish the set, *R*.
  - 3. For each $\mathbf{d}_j \in R$, calculate *Similarity*(**q**, $\mathbf{d}_j$), using appropriate weights.
  - 4. Return the elements of *R* in ranked order.

Inverted Index With Counts

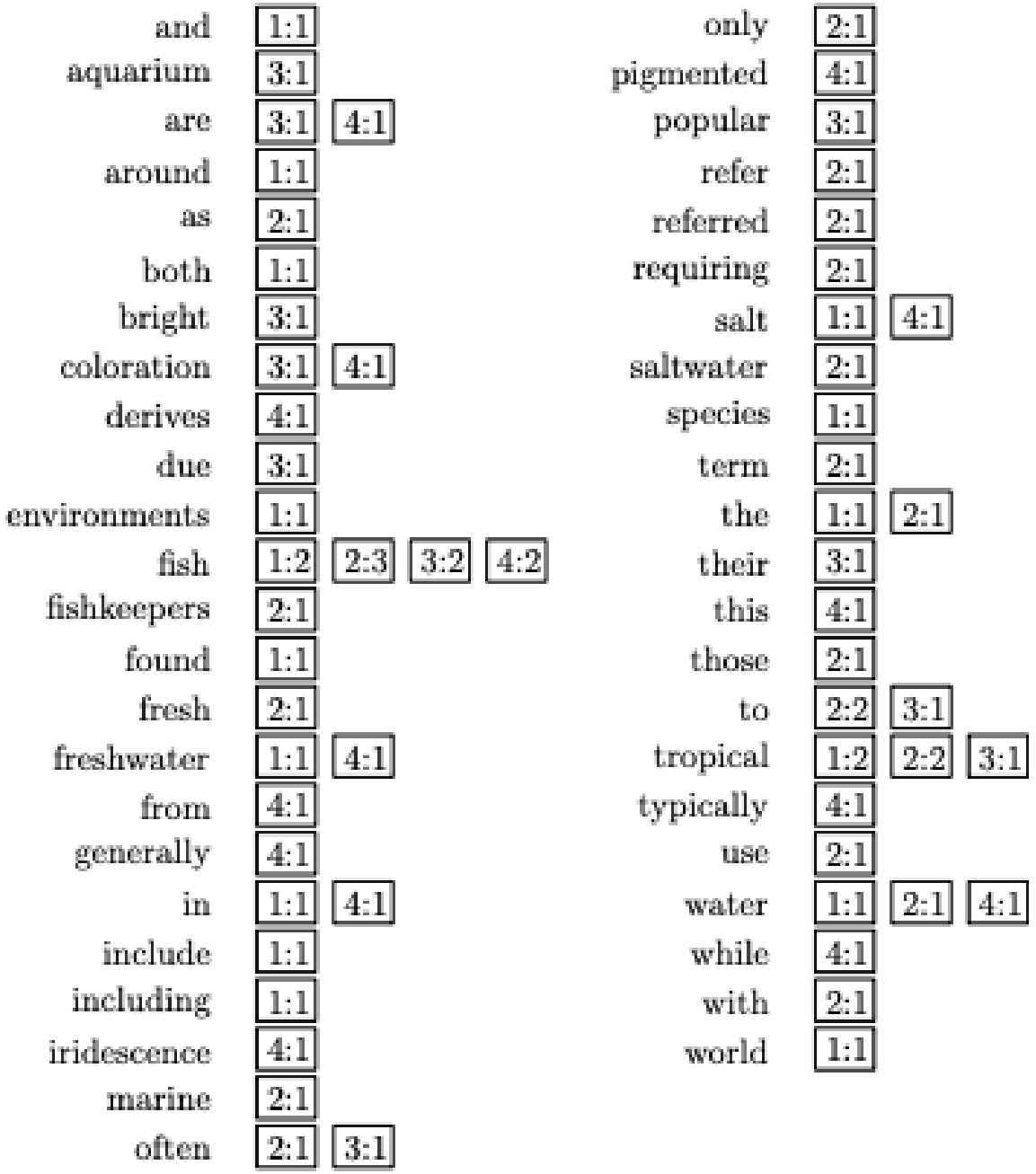
- In the previous example, terms were binary, 1 if the term is in the document and 0 otherwise.
  - Assume the query that contains the term “tropical”
  - An equal score will be generated for S_1,S_2,S_3
  - But S_2 should have a higher rank given that “tropical” appears twice in the document
- Lets update the index to contain frequency information

Example – Inverted Index With Counts



Postings File

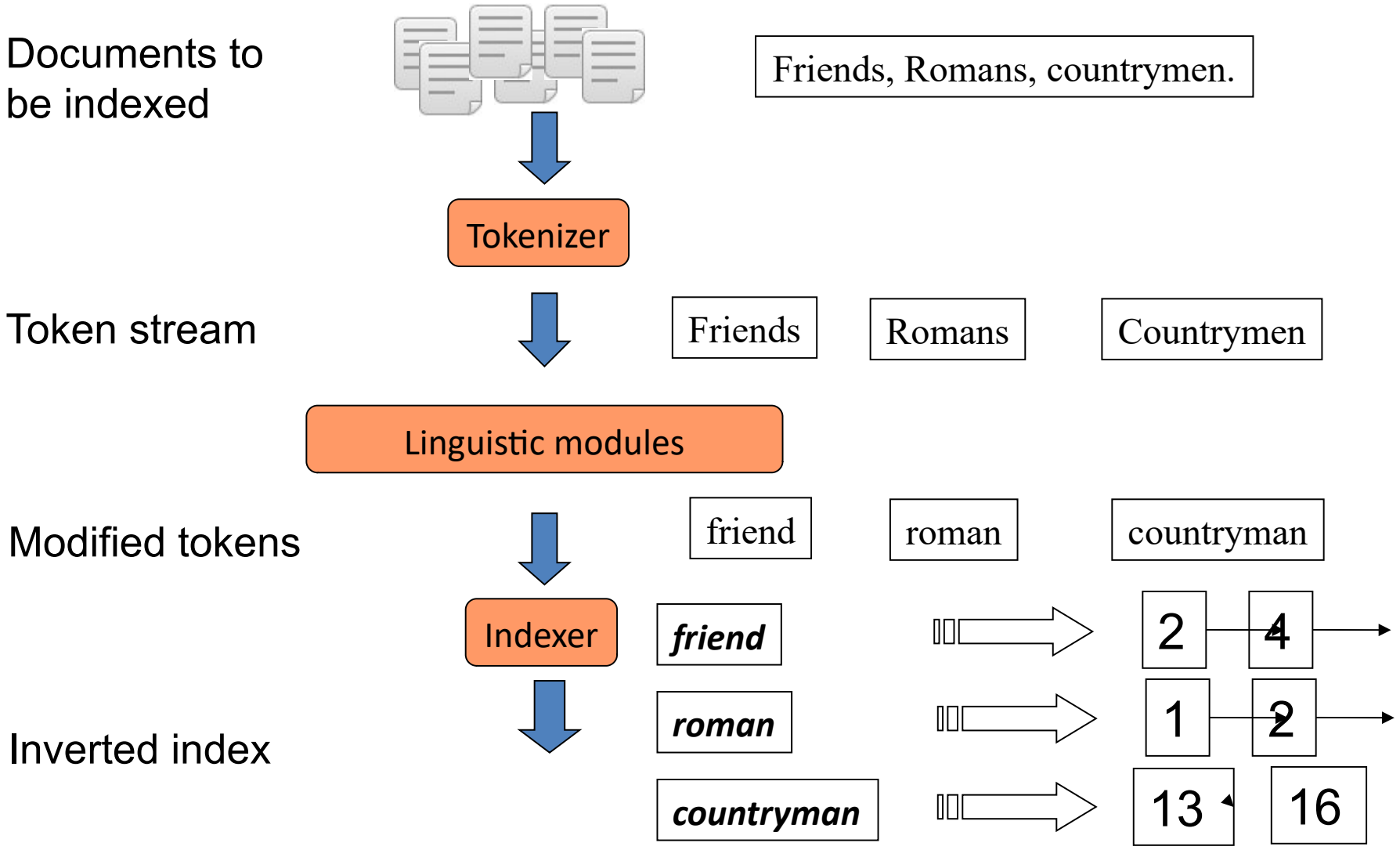
- The **postings file** stores the elements of a sparse matrix, the term assignment matrix, with weights.
- It is stored as a separate **inverted list** for each column, i.e., a list corresponding to each term in the index file.
- Each element in an inverted list is called a **posting**, i.e., the occurrence of a term in a document
- Each list consists of one or many individual postings.



Postings File

- Merging inverted lists is the most computationally intensive task in many information retrieval systems.
- Since inverted lists may be long, it is important to match postings efficiently.
- *Usually, the inverted lists will be held on disk and paged into memory for matching*. Therefore algorithms for matching postings process the lists sequentially.
- For efficient matching, the inverted lists should all be sorted in the same sequence.
- Inverted lists are commonly cached to minimize disk accesses.

Inverted index construction



Initial stages of text processing

- Tokenization
  - Cut character sequence into word tokens
    - Deal with “John’s”, a state-of-the-art solution
- Normalization
  - Map text and query term to same form
    - You want U.S.A. and USA to match
- Stemming
  - We may wish different forms of a root to match
    - authorize, authorization
- Stop words
  - We may omit very common words (or not)
    - the, a, to, of

Indexer steps: Token sequence

- Sequence of (Modified token, Document ID) pairs.

Doc 1

I did enact Julius Caesar I was killed i' the Capitol; Brutus killed me.

Doc 2

So let it be with Caesar. The noble Brutus hath told you Caesar was ambitious

Term	docID
I	1
did	1
enact	1
julius	1
caesar	1
I	1
was	1
killed	1
i'	1
the	1
capitol	1
brutus	1
killed	1
me	1
so	2
let	2
it	2
be	2
with	2
caesar	2
the	2
noble	2
brutus	2
hath	2
told	2
you	2
caesar	2
was	2
ambitious	2

Indexer steps: Sort

- Sort by terms
  - And then docID

Term	docID
I	1
did	1
enact	1
julius	1
caesar	1
I	1
was	1
killed	1
i'	1
the	1
capitol	1
brutus	1
killed	1
me	1
so	2
let	2
it	2
be	2
with	2
caesar	2
the	2
noble	2
brutus	2
hath	2
told	2
you	2
caesar	2
was	2
ambitious	2

Term	docID
ambitious	2
be	2
brutus	1
brutus	2
capitol	1
caesar	1
caesar	2
caesar	2
did	1
enact	1
hath	1
I	1
I	1
i'	1
it	2
julius	1
killed	1
killed	1
let	2
me	1
noble	2
so	2
the	1
the	2
told	2
you	2
was	1
was	2
with	2

Indexer steps: Dictionary & Postings

- Multiple term entries in a single document are merged.
- Split into Dictionary and Postings
- Term document frequency information is added.
  - # document in which term shows up
  - Is used for TF-IDF computation
  - Also, used for query optimization

- Query optimization Example
  - Assume binary model and doc frequency given as
    - Caesar =100
    - Julius = 50
    - Ambititous = 5

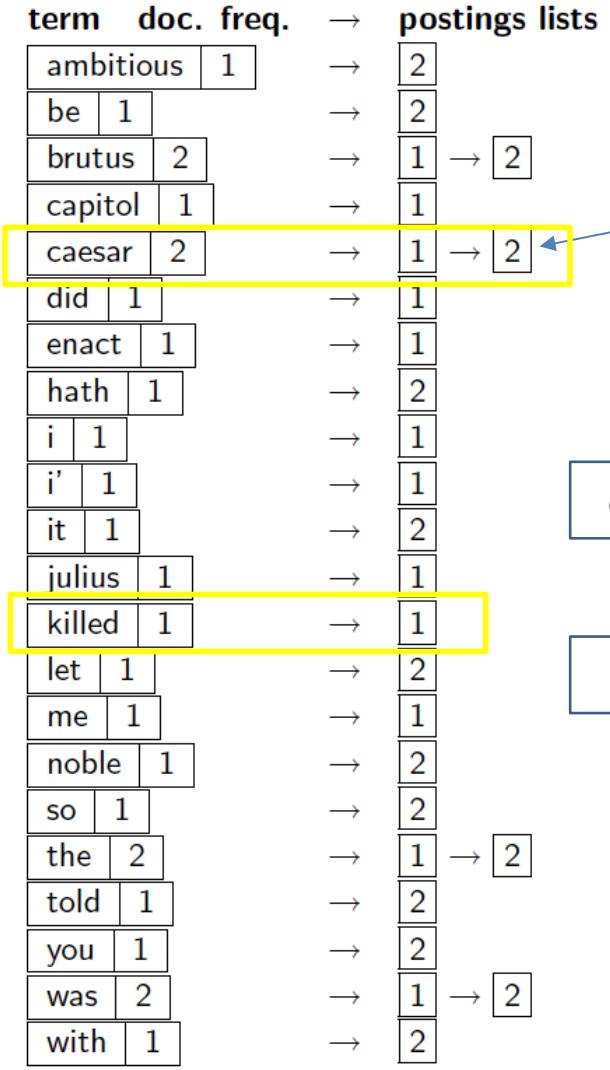
    - Is this optimal?
      - Caesar AND Julius AND Ambitious

Term	docID
ambitious	2
be	2
brutus	1
brutus	2
capitol	1
caesar	1
caesar	2
caesar	2
did	1
enact	1
hath	1
I	1
I	1
i'	1
it	2
julius	1
killed	1
killed	1
let	2
me	1
noble	2
so	2
the	1
the	2
told	2
you	2
was	1
was	2
with	2

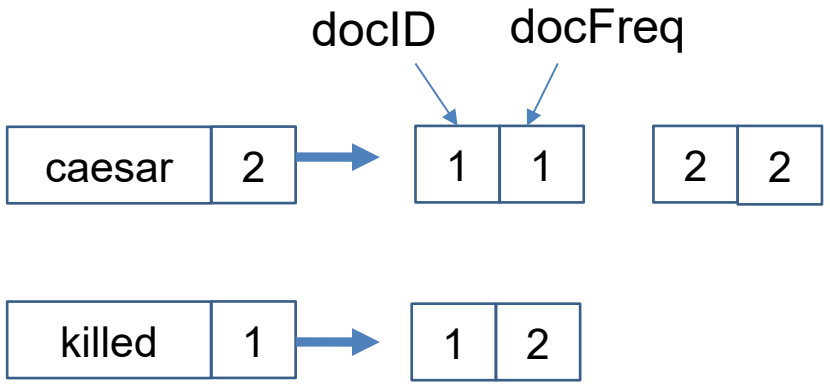
term	doc. freq.	→	postings lists
ambitious	1	→	2
be	1	→	2
brutus	2	→	1 → 2
capitol	1	→	1
caesar	2	→	1 → 2
did	1	→	1
enact	1	→	1
hath	1	→	2
i	1	→	1
i'	1	→	1
it	1	→	2
julius	1	→	1
killed	1	→	1
let	1	→	2
me	1	→	1
noble	1	→	2
so	1	→	2
the	2	→	1 → 2
told	1	→	2
you	1	→	2
was	2	→	1 → 2
with	1	→	2

Indexer steps: Dictionary & Postings (with frequency info)

- Add frequency of term within a given document.



Augment postings to contain the frequency within a given document



Additional Indices

- Document Index

DocID	Length
1	432
2	13432
3	500
4	10982
5	1000

This is used conjunction with the posting list to compute TF-IDF weights

Phrase queries

- We want to be able to answer queries such as ***“Tropical Fish”*** – as a phrase
- Thus the sentence *”I would love to vacation somewhere tropical, and eat coconuts and fresh fish ”* is not a match.
  - The concept of phrase queries has proven easily understood by users; one of the few “advanced search” ideas that works
  - Many more queries are *implicit phrase queries*
- For this, it no longer suffices to store only <*term* : *docs*> entries

A first attempt: Biword indexes

- Index every consecutive pair of terms in the text as a phrase
- For example the text “Friends, Romans, Countrymen” would generate the biwords
  - ***friends romans***
  - ***romans countrymen***
- Each of these biwords is now a dictionary term
- Two-word phrase query-processing is now immediate.

Longer phrase queries

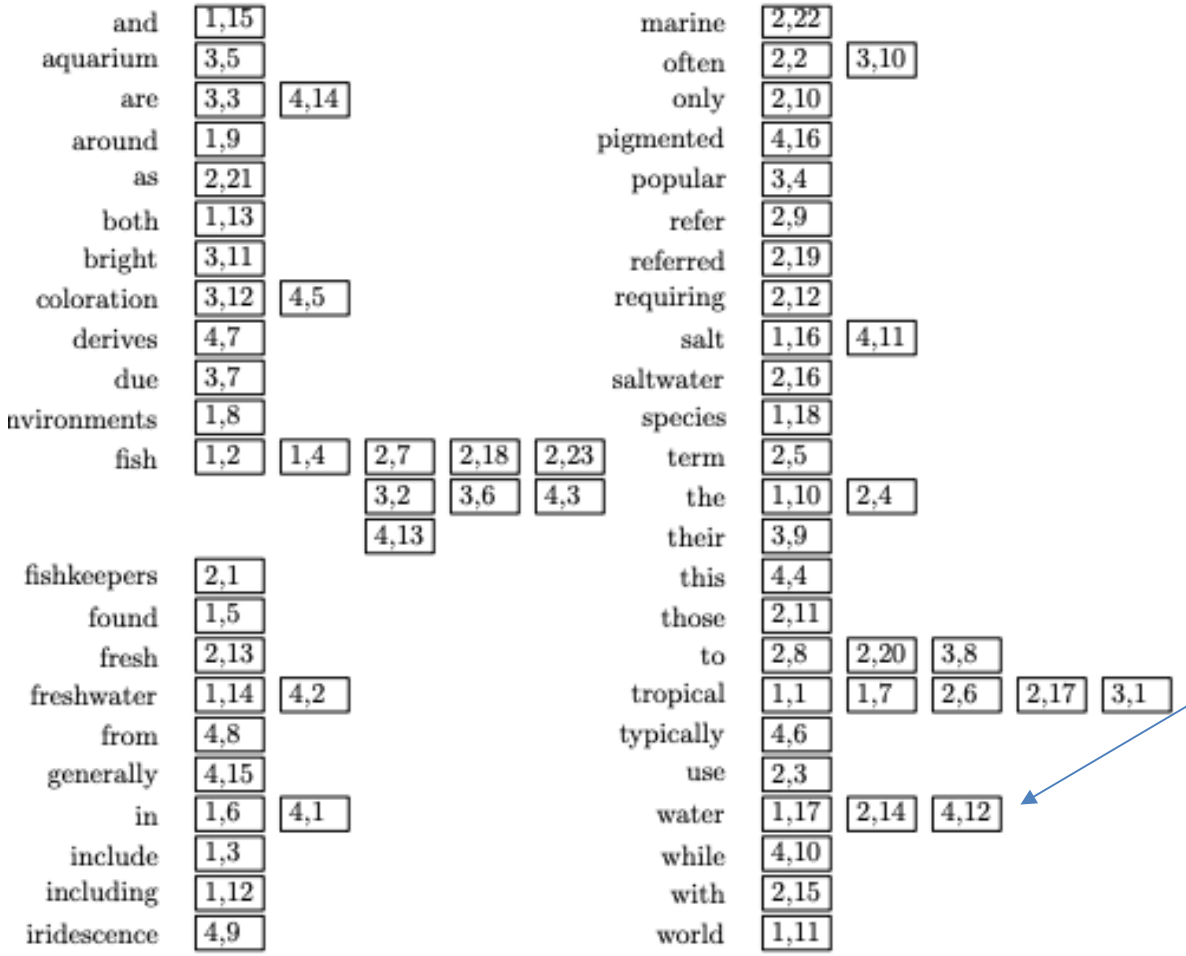
- Longer phrases can be processed by breaking them down
- ***Example:***

Yellow and green tropical fish can be broken into the Boolean query on biwords

Yellow green *AND* ***green tropical*** *AND* ***tropical fish***

- Disadvantages
  - Without the docs, we cannot verify that the docs matching the above Boolean query does contain the exact phrase.
    - Will have a lot of false positives
  - Index blowup due to bigger dictionary
    - Infeasible for more than b-iwords (the index will be huge even if we just consider biwords)

Example – Inverted Index With Positions



- In the postings, store, for each ***term*** the position(s) in which tokens of it appear:

<***term***, number of docs containing ***term***;
doc1: position1, position2 … ;
doc2: position1, position2 … ;
etc. >

Postings contain the document id and the position of the word in the document.

(document id, position of term in document)

Evaluating Queries

- Given a query Q = “Tropical Fish”
  - We first find the relevant posting of each term and align the posting entries

  - We then examine the intersection between the two lists
  - And finally determine if it contains an exact match of “tropical fish” based on the position information
  - Positional information has an added benefit of allowing us to highlight the position of phrases/terms in the document!

What about proximity queries?

- Q = Tropical /2 Fish
  - where /*k* means “within *k* words of”.

- Clearly, positional indexes can be used for such queries; bi-word indexes cannot.

- Take home problem
  - Adapt the linear merge of postings to handle proximity queries. Can you make it work for any value of *k*?
    - This is a little tricky to do correctly and efficiently

Positional index size

- Need an entry for each occurrence, not just once per document
- Index size depends on average document size
  - Average web page has <1000 terms
  - SEC filings, books, even some epic poems … easily 100,000 terms
- Consider a term with frequency 0.1%

Document Size	Postings	Positional Postings
1000	1	1
100,1000	1	100

- Rule of thumb
  - A positional index is 2–4 as large as a non-positional index
  - Positional index size 35–50% of volume of original text
    - Caveat: all of this holds for “English-like” languages

Combination schemes

- These two approaches can be profitably combined
  - For particular phrases (***"Michael Jackson", "Britney Spears"***) it is inefficient to keep on merging positional postings lists
    - Even more so for phrases like ***"The Who"***

- "Fast Phrase Querying with Combined Indexes" by Williams et al. (2004) evaluate a more sophisticated mixed indexing scheme
  - A typical web query mixture was executed in ¼ of the time of using just a positional index
  - It required 26% more space than having a positional index alone

Three Laws of Text

- Word occurrences governed by statistical laws:

- **Zipf's law :** frequency and rarity of words
- **Heaps law:** rate at which new words will appear
- **Dependence:** clumpy / contagious nature of words

Zipf's Law

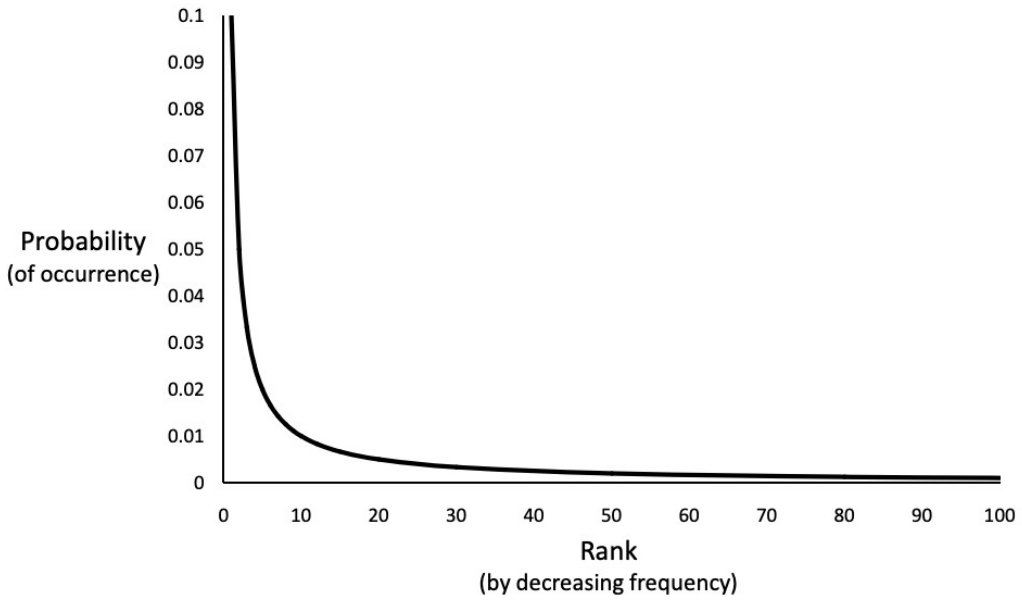
- Observation - there are frequent words and rare words
  - words "of" and "the" make up 10% of all occurrences
- Rank words by frequency
- Zipf's law
  - Models the relative popularity of a few words and the relative obscurity of other words
  - Also applied to other examples
    - Website
    - library books
    - population by city

Word	Freq.	r	Pr(%)
the	2,420,778	1	6.49
of	1,045,733	2	2.90
to	968,882	3	2.60
a	892,429	4	2.39
and	865,644	5	2.32
in	847,825	6	2.27

$$r * Pr \cong c \text{ (constant)}$$

$$c = 0.01 \text{ for English corpus}$$

AP89 Collection

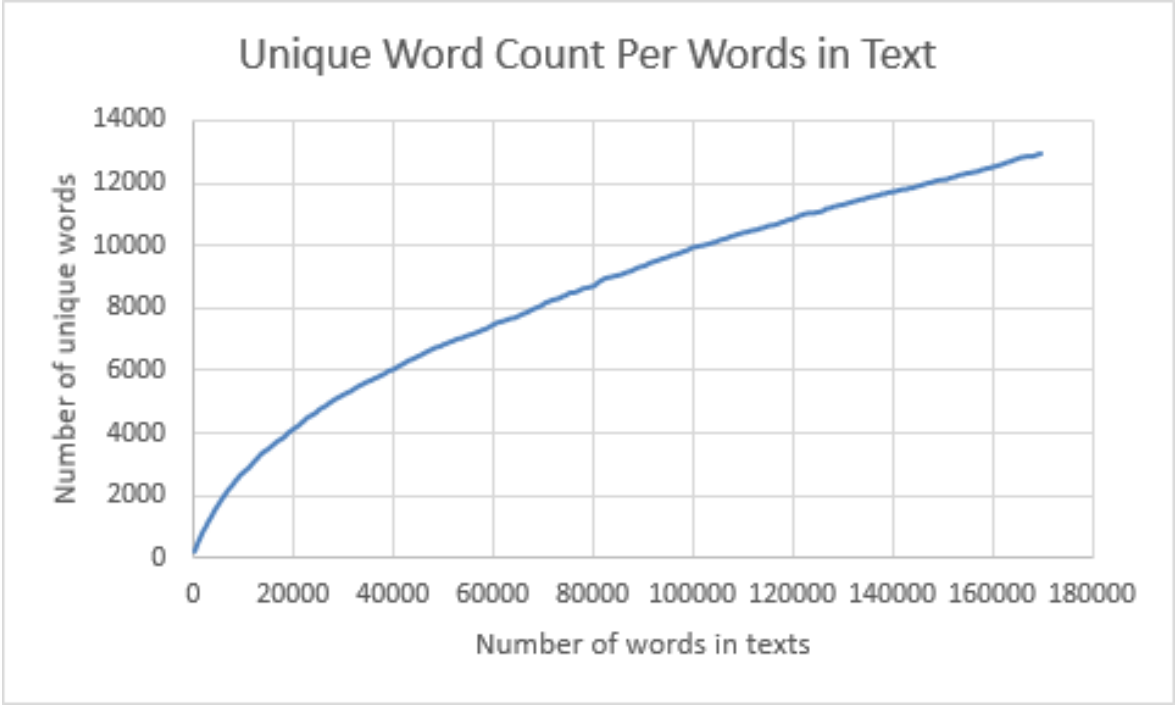


Word	*Freq.*	*r*	P_r(%)	$r.P_r$	*Word*	*Freq*	*r*	P_r(%)	$r.P_r$
the	2,420,778	1	6.49	0.065	has	136,007	26	0.37	0.095
of	1,045,733	2	2.80	0.056	are	130,322	27	0.35	0.094
to	968,882	3	2.60	0.078	not	127,493	28	0.34	0.096
a	892,429	4	2.39	0.096	who	116,364	29	0.31	0.090
and	865,644	5	2.32	0.120	they	111,024	30	0.30	0.089
in	847,825	6	2.27	0.140	its	111,021	31	0.30	0.092
said	504,593	7	1.35	0.095	had	103,943	32	0.28	0.089
for	363,865	8	0.98	0.078	will	102,949	33	0.28	0.091
that	347,072	9	0.93	0.084	would	99,503	34	0.27	0.091
was	293,027	10	0.79	0.079	about	92,983	35	0.25	0.087
on	291,947	11	0.78	0.086	i	92,005	36	0.25	0.089
he	250,919	12	0.67	0.081	been	88,786	37	0.24	0.088
is	245,843	13	0.65	0.086	this	87,286	38	0.23	0.089
with	223,846	14	0.60	0.084	their	84,638	39	0.23	0.089
at	210,064	15	0.56	0.085	new	83,449	40	0.22	0.090
by	209,586	16	0.56	0.090	or	81,796	41	0.22	0.090
it	195,621	17	0.52	0.089	which	80,385	42	0.22	0.091
from	189,451	18	0.51	0.091	we	80,245	43	0.22	0.093
as	181,714	19	0.49	0.093	more	76,388	44	0.21	0.090
be	157,300	20	0.42	0.084	after	75,165	45	0.20	0.091
were	153,913	21	0.41	0.087	us	72,045	46	0.19	0.089
an	152,576	22	0.41	0.090	percent	71,956	47	0.19	0.091
have	149,749	23	0.40	0.092	up	71,082	48	0.19	0.092
his	142,285	24	0.38	0.092	one	70,266	49	0.19	0.092
but	140,880	25	0.38	0.094	people	68,988	50	0.19	0.093

Heap's Law

- **Experiment:** read a book / newspaper / website
  - record every time you see a new word
  - v = number of new words seen, n = total words
  - plot v against n

- **Vocabulary growth**
- $v = k \times n^b$
- b usually equal to .5



Clumping / Contagion

- Word occurrences: rare but “contagious” events
  - a-priori, you’re very unlikely to see a give word
  - but, see it once, you are much more likely to see it again

Applying the laws … why we care

- Assume you are given a 200GB crawl of English web pages
- What can you guess or approximate about the index size

- Heap's Law
  - about 35 unique index terms (and about 20b total words when assuming an approximate word length)

- Zipf's Law
  - about 17m will have only one occurrence, and the word "the" will have > 10m

- clumping
  - how often we should expect > 1 entry for rare words