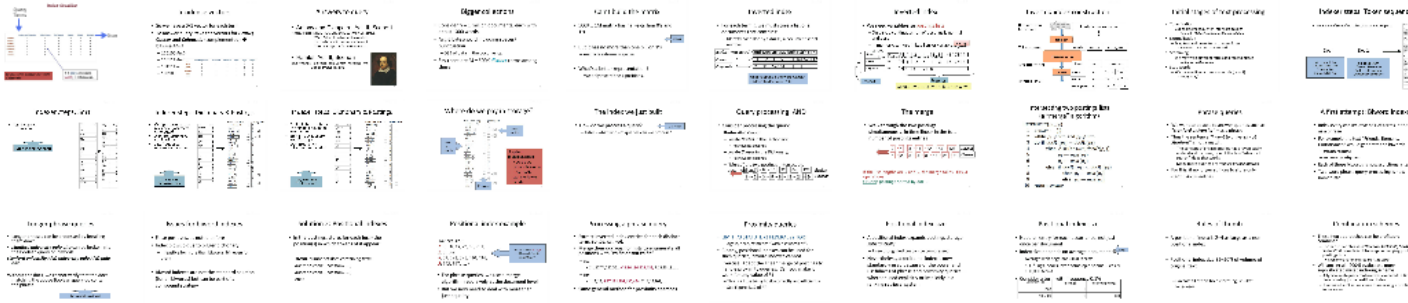


Web Page Indexing: TF-IDF

Problem- Definition



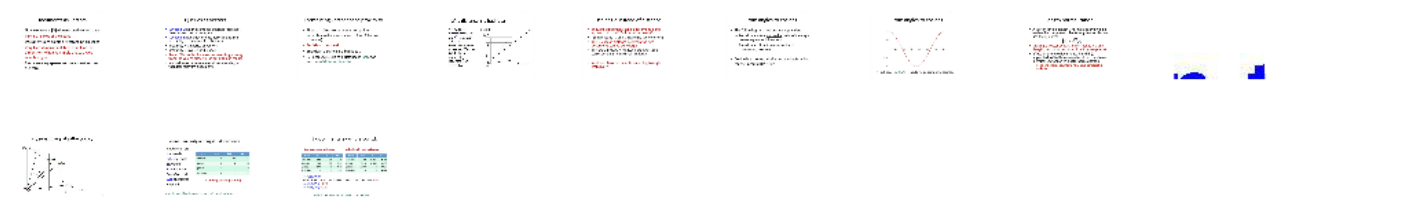
Indexing



Ranked Retrieval



Vector Space Model



Web Page Indexing: TF-IDF

Problem- Definition

Problem-Definition

- Problem Statement: How can we efficiently index and retrieve information from a large corpus of web pages?
- Key Challenges:
 - Volume: Handling billions of documents.
 - Sparsity: Many terms are unique to a single document.
 - Relevance: Identifying the most important terms.
 - Efficiency: Storing and searching the index quickly.

Indexing

Indexing

- Document Representation: Converting text into a numerical vector.
- Term Frequency (TF): The number of times a term appears in a document.
- Inverse Document Frequency (IDF): A measure of how important a term is, based on how many documents it appears in.
- TF-IDF: The product of TF and IDF, representing the weight of a term in a document.
- Indexing Process: Building an inverted index where each term points to the documents it appears in.

Ranked Retrieval

Ranked Retrieval

- Query Processing: Analyzing the user's search query.
- Scoring: Calculating the relevance score for each document in the index.
- Ranking: Ordering the documents based on their scores.
- Retrieval: Returning the top-ranked documents to the user.

Vector Space Model

Vector Space Model

- Document as Vectors: Representing documents as points in a high-dimensional space.
- Similarity Measures: Calculating the cosine similarity between document vectors.
- Dimensionality Reduction: Using techniques like PCA or SVD to reduce the number of dimensions.

Information Retrieval (IR) - The Problem



Collection: A set of documents Assume it is a static collection for the moment

Goal: Retrieve documents with information that is relevant to the user's information need and helps the user complete a task

Searching is a Solved Problem!

Hash Table: $O(1)$

Binary Search Tree: $O(\log(n))$

AVL Tree: $O(\log(n))$

.....

Why Do we need a different setup?



Size of the Data!

- 10 billion web pages
- Average size of webpage = 20KB
- 10 billion * 20KB = 200 TB
- Disk read bandwidth = 50 MB/sec
- Time to read = 4 million seconds = 46+ days
- Say there are $M = 500K$ distinct terms among these.
- Even longer to do something useful with the data

Dictionary - Desired DS

A naïve dictionary

- An array of struct:

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

char[20] int Postings *

20 bytes 4/8 bytes 4/8 bytes

- How do we store a dictionary in memory efficiently?
- How do we quickly look up elements at query time?

Dictionary - Desired DS

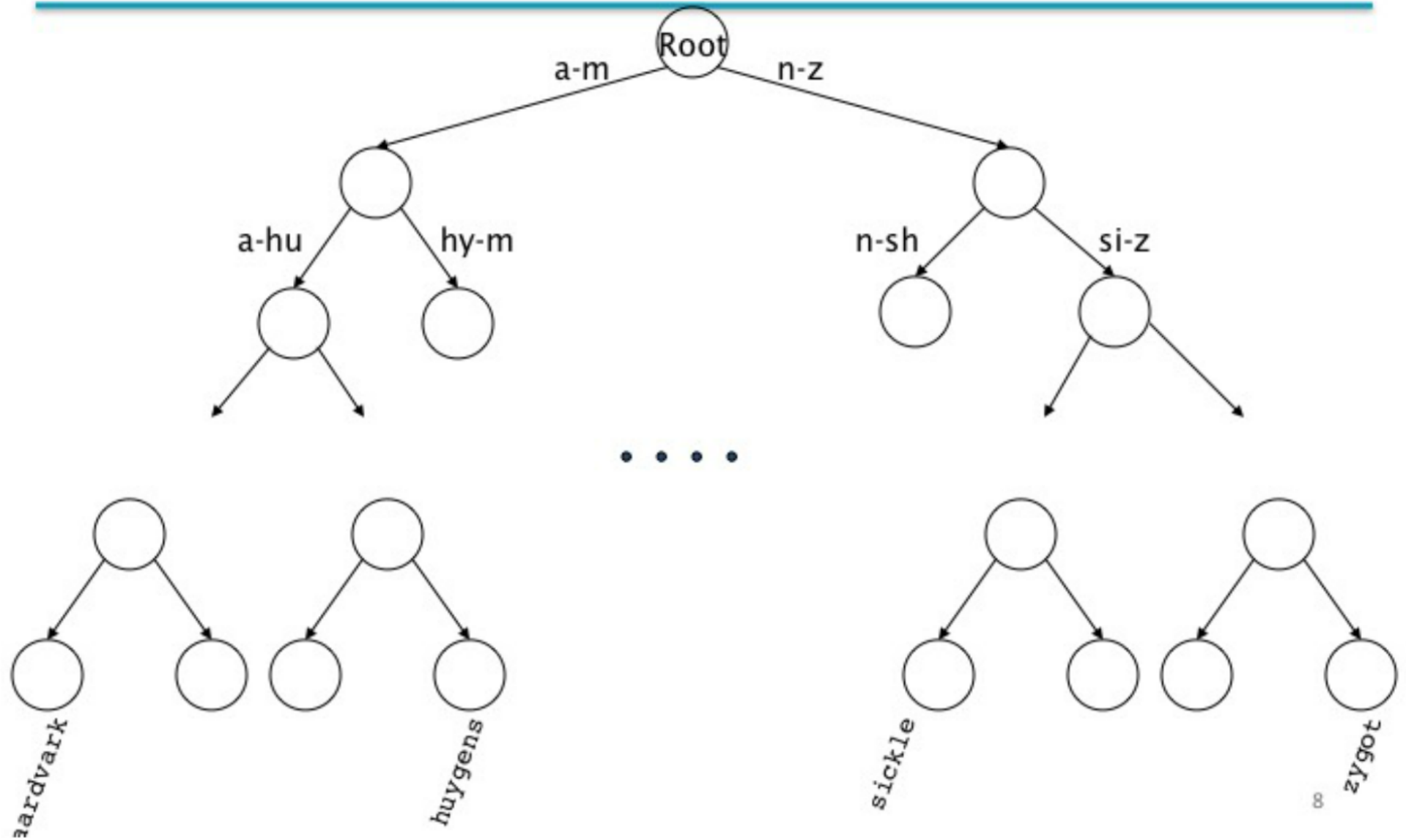
Dictionary data structures

- Two main choices:
 - Hashtables
 - 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



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

Real Challenges

- Out-of-Vocabulary Words
 - 20-25% Named Entities
- Multilingual Queries
- Wild Card Queries
- Spell Correction
- Phrase Search
 - President of India*

Web Page Indexing: TF-IDF

Problem- Definition

Information Retrieval (IR): The Problem

Collection: A set of documents assembled in a public collection for the purposes of retrieval.

Goal: Retrieve documents which information that is relevant to the user's information need and helps the user complete a task.

Searching is a Solvable Problem!

Highly-Structured (e.g. Web Search, Tree, Graph) vs. Low-Structured (e.g. Text, Graph)

Why do we need a different setup?

Size of the Data!

- 10 billion webpages
- 10 billion unique words
- 10 billion unique documents
- 10 billion unique documents
- 10 billion unique documents
- 10 billion unique documents
- 10 billion unique documents
- 10 billion unique documents
- 10 billion unique documents
- 10 billion unique documents

Indexing - Desired DS

Dictionary data structures for inverted indexes

Indexing - Desired DS

Dictionary - Desired DS

Dictionary data structures

Indexing - Desired DS

Indexing - Desired DS

Indexing - Desired DS

Indexing - Desired DS

Indexing - Desired DS

Indexing - Desired DS

Indexing - Desired DS

Indexing - Desired DS

Indexing - Desired DS

Indexing - Desired DS

Indexing - Desired DS

Indexing

Indexing - Desired DS

Indexing - Desired DS

Indexing - Desired DS

Indexing - Desired DS

Indexing - Desired DS

Indexing - Desired DS

Indexing - Desired DS

Indexing - Desired DS

Indexing - Desired DS

Indexing - Desired DS

Indexing - Desired DS

Indexing - Desired DS

Indexing - Desired DS

Indexing - Desired DS

Indexing - Desired DS

Indexing - Desired DS

Indexing - Desired DS

Indexing - Desired DS

Indexing - Desired DS

Indexing - Desired DS

Ranked Retrieval

Ranked retrieval

Ranked retrieval

Ranked retrieval models

Ranked retrieval models

Ranked retrieval models

Ranked retrieval models

Ranked retrieval models

Ranked retrieval models

Ranked retrieval models

Ranked retrieval models

Ranked retrieval models

Ranked retrieval models

Ranked retrieval models

Ranked retrieval models

Ranked retrieval models

Ranked retrieval models

Ranked retrieval models

Ranked retrieval models

Ranked retrieval models

Ranked retrieval models

Index Creation

Query
Terms



	Antony and Cleopatra	Julius Caesar	The Tempest	Hamlet	Othello	Macbeth
Antony	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

Docs

***Brutus AND Caesar BUT NOT
Calpurnia***

1 if play contains
word, 0 otherwise

Incidence vectors

- So we have a 0/1 vector for each term.
- To answer query: take the vectors for **Brutus**, **Caesar** and **Calpurnia** (complemented) → bitwise **AND**.

– 110100 **AND**

– 110111 **AND**

– 101111 =

– **100100**

	Antony and Cleopatra	Julius Caesar	The Tempest	Hamlet	Othello	Macbeth
Antony	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

Answers to query

- **Antony and Cleopatra, Act III, Scene ii**

Agrippa [Aside to DOMITIUS ENOBARBUS]: Why, Enobarbus,
When Antony found Julius **Caesar** dead,
He cried almost to roaring; and he wept
When at Philippi he found **Brutus** slain.

- **Hamlet, Act III, Scene ii**

Lord Polonius: I did enact Julius **Caesar** I was killed i' the
Capitol; **Brutus** killed me.

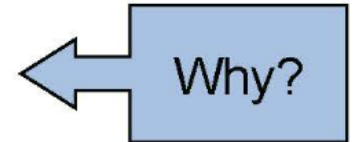


Bigger collections

- Consider $N = 1$ million documents, each with about 1000 words.
- Avg 6 bytes/word including spaces/punctuation
 - 6GB of data in the documents.
- Say there are $M = 500K$ *distinct* terms among these.

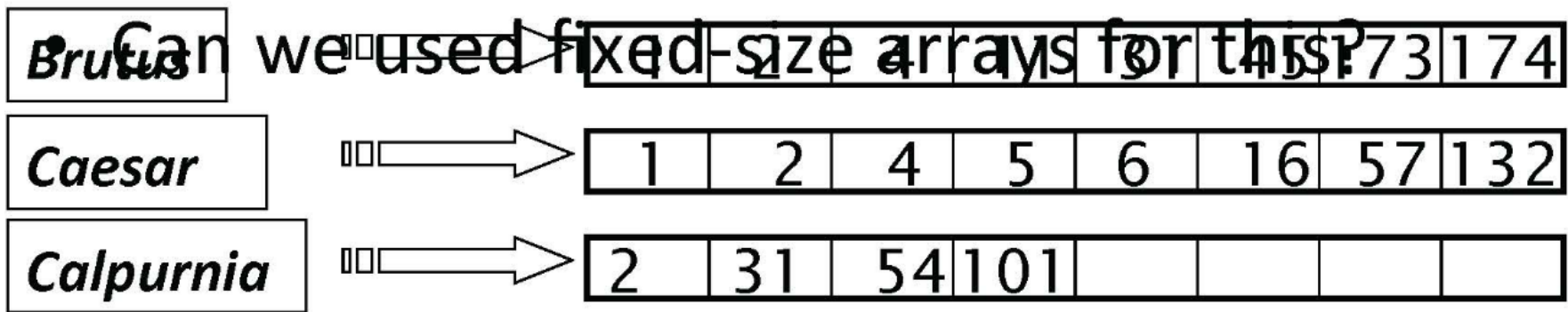
Can't build the matrix

- 500K x 1M matrix has half-a-trillion 0's and 1's.
- But it has no more than one billion 1's.
 - matrix is extremely sparse.
- What's a better representation?
 - We only record the 1 positions.



Inverted index

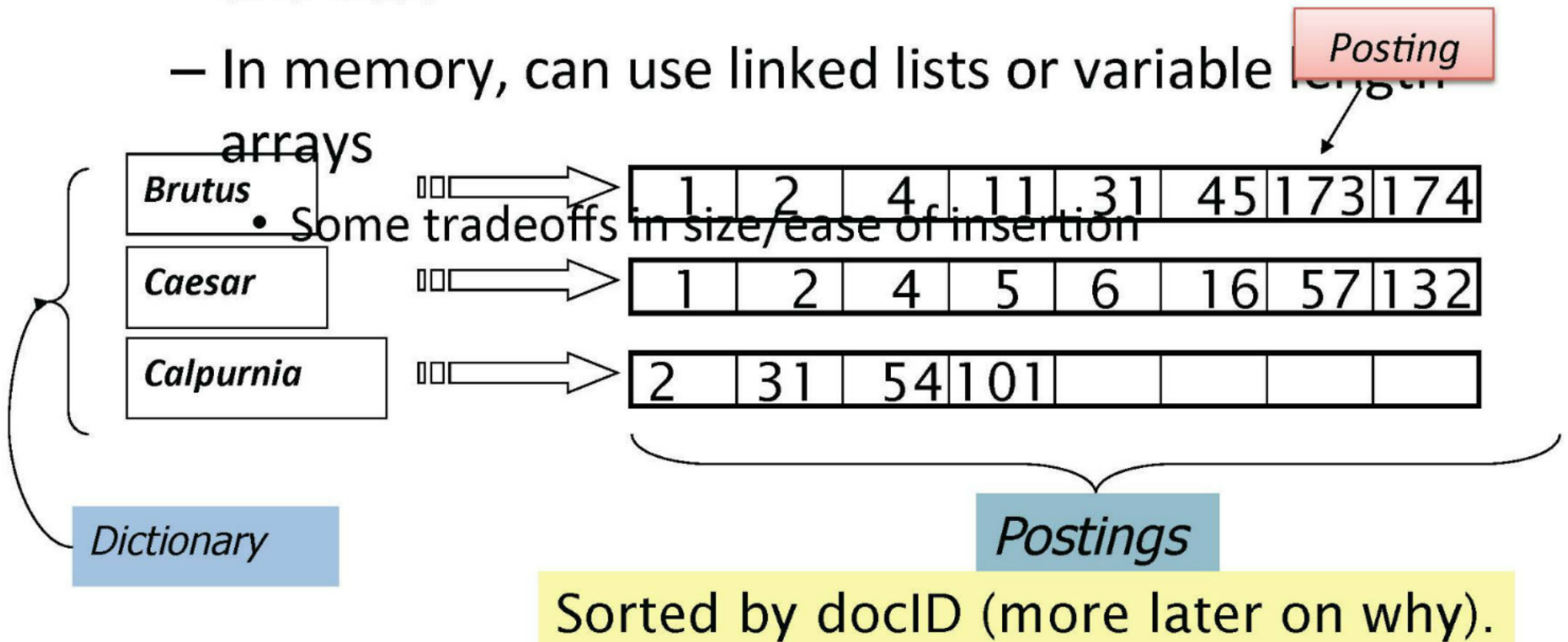
- For each term t , we must store a list of all documents that contain t .
 - Identify each doc by a **docID**, a document serial number



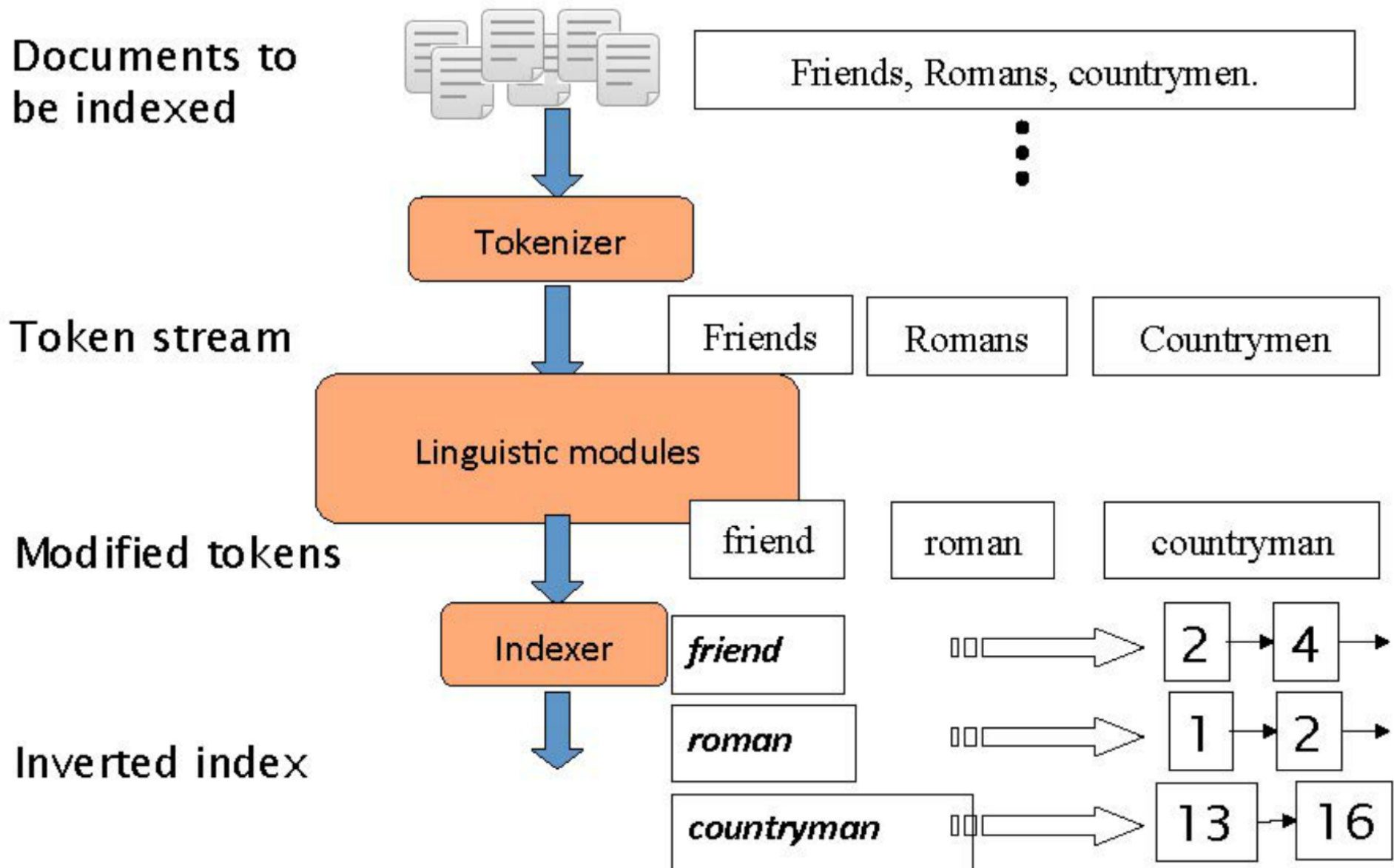
What happens if the word *Caesar* is added to document 14?

Inverted index

- We need variable-size **postings lists**
 - On disk, a continuous run of postings is normal and best
 - In memory, can use linked lists or variable length



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

Core indexing step

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
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
- Doc. frequency information is added.

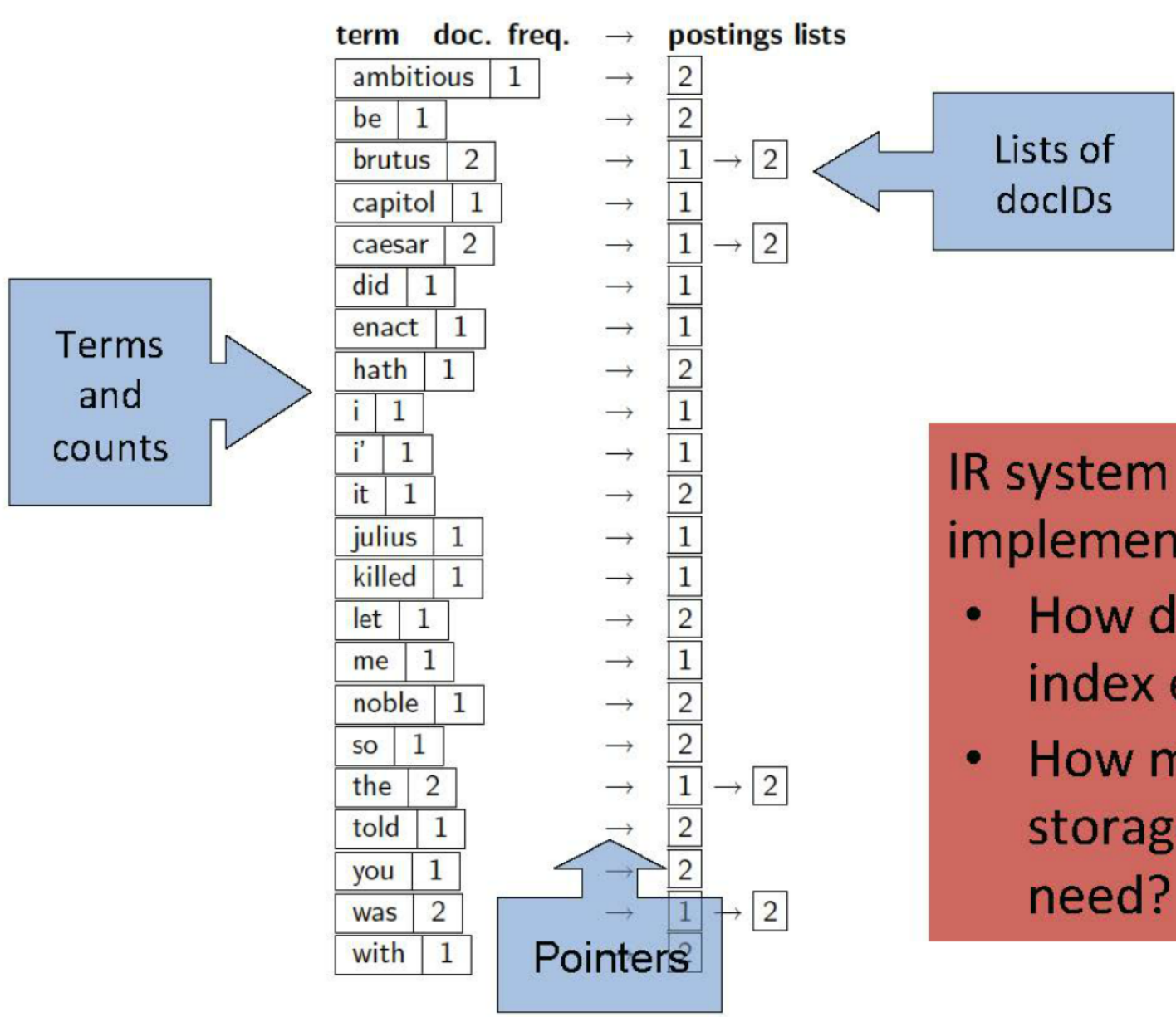
Term	docID
ambitious	2
be	2
brutus	1
brutus	2
capitol	1
caesar	1
caesar	2
caesar	2
did	1
enact	1
hath	1
I	1
I	1
i'	1
it	2
julius	1
killed	1
killed	1
let	2
me	1
noble	2
so	2
the	1
the	2
told	2
you	2
was	1
was	2
with	2



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]

Why frequency?
Will discuss later.


Where do we pay in storage?



IR system implementation

- How do we index efficiently?
- How much storage do we need?

The index we just built

- How do we process a query?
 - Later - what kinds of queries can we process?

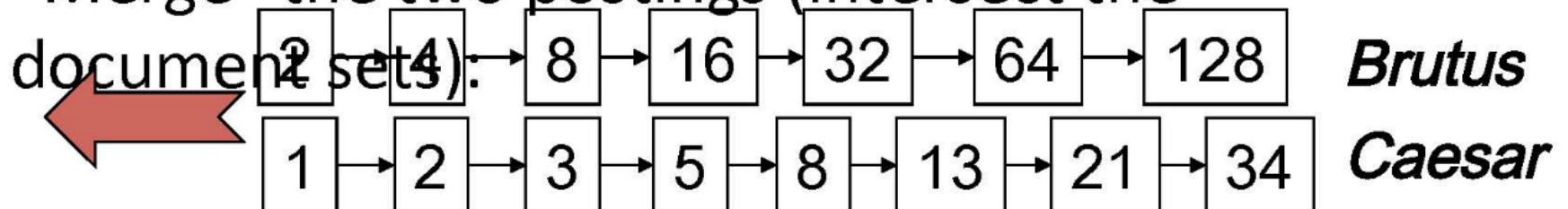
Query processing: AND

- Consider processing the query:

Brutus AND Caesar

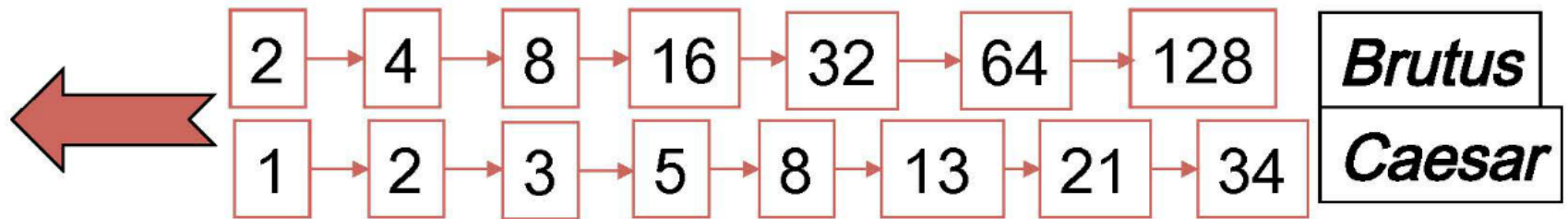
- Locate *Brutus* in the Dictionary;
 - Retrieve its postings.
- Locate *Caesar* in the Dictionary;
 - Retrieve its postings.

- “Merge” the two postings (intersect the



The merge

- Walk through the two postings simultaneously, in time linear in the total number of postings entries



If the list lengths are x and y , the merge takes $O(x+y)$ operations.

Crucial: postings sorted by docID.

Phrase queries

- We want to be able to answer queries such as “***stanford university***” – as a phrase
- Thus the sentence “*I went to university at Stanford*” 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
- ***stanford university palo alto*** can be broken into the Boolean query on biwords:

stanford university AND university palo AND palo alto

Without the docs, we cannot verify that the docs matching the above Boolean query do contain the phrase.



Can have false positives!

Issues for biword indexes

- False positives, as noted before
- Index blowup due to bigger dictionary
 - Infeasible for more than biwords, big even for them
- Biword indexes are not the standard solution (for all biwords) but can be part of a compound strategy

Solution 2: Positional indexes

- 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.>

Positional index example

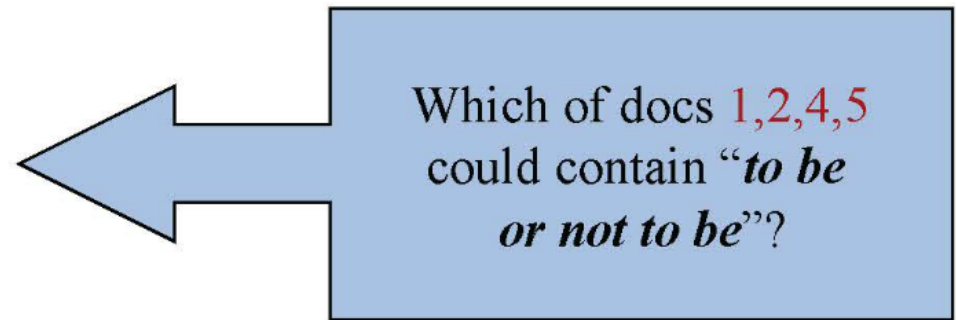
<*be*: 993427;

1: 7, 18, 33, 72, 86, 231;

2: 3, 149;

4: 17, 191, 291, 430, 434;

5: 363, 367, ...>



- For phrase queries, we use a merge algorithm recursively at the document level
- But we now need to deal with more than just equality

Processing a phrase query

- Extract inverted index entries for each distinct term: ***to, be, or, not.***
- Merge their *doc:position* lists to enumerate all positions with “***to be or not to be***”.
 - ***to:***
 - 2:1,17,74,222,551; **4:8,16,190,429,433**; 7:13,23,191; ...
 - ***be:***
 - 1:17,19; **4:17,191,291,430,434**; 5:14,19,101; ...
- Same general method for proximity searches

Proximity queries

- **LIMIT! /3 STATUTE /3 FEDERAL /2 TORT**
 - Again, here, / k means “within k words of”.
- Clearly, positional indexes can be used for such queries; biword indexes cannot.
- Exercise: 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
 - See Figure 2.12 of *IIR*

Positional index size

- A positional index expands postings storage *substantially*
 - Even though indices can be compressed
- Nevertheless, a positional index is now standardly used because of the power and usefulness of phrase and proximity queries ... whether used explicitly or implicitly in a ranking retrieval system.

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
100,000	1	100

Rules 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”***
- Williams et al. (2004) evaluate a more sophisticated mixed indexing scheme
 - A typical web query mixture was executed in $\frac{1}{4}$ of the time of using just a positional index
 - It required 26% more space than having a positional index alone

Incidence sources

- Flowchart showing data flow from sources to processing.

Incidence sources (continued)

- Diagram of data flow.

Longer phrase queries

- Diagram showing query expansion.

Analysis to query

- Diagram of query analysis.

Incidence sources: frequency & ratings

- Diagram of frequency and ratings.

Solution 2: Positional indexes

- Diagram of positional indexing.

Bigger collections

- Text about larger collections.

Where do we stay in storage

- Diagram of storage locations.

Positional Index estimate

- Diagram of index estimation.

Can't build the matrix

- Text about matrix building.

The index we just built

- Text about the built index.

Processing a phrase query

- Text about phrase query processing.

Inverted Index

- Diagram of inverted index.

Query processing: AND

- Text about AND query processing.

Proximity queries

- Text about proximity queries.

Initial stages of text processing

- Diagram of text processing stages.

Private queries

- Text about private queries.

Rules of thumb

- Text about rules of thumb.

A first estimate: Inward indexes

- Text about inward indexes.

Combination schemes

- Text about combination schemes.

Positional Index size

- Text about positional index size.

Positional Index size (continued)

- Text about positional index size.

Positional Index size

- Text about positional index size.

Positional Index size (continued)

- Text about positional index size.

Positional Index size

- Text about positional index size.

Positional Index size (continued)

- Text about positional index size.

Ranked Retrieval

Ranked retrieval

- Text about ranked retrieval.

Problem with Boolean search

- Text about Boolean search problems.

Ranked retrieval models

- Text about ranked retrieval models.

Term frequency

- Text about term frequency.

Feist or formula: not a problem in ranked retrieval

- Text about Feist or formula.

Log frequency weighting

- Text about log frequency weighting.

Scoring as the basis of ranked retrieval

- Text about scoring as the basis.

Document frequency

- Text about document frequency.

Query-document matching scores

- Text about matching scores.

Document frequency, continued

- Text about document frequency.

Term-document count matrices

- Text about count matrices.

log of words model

- Text about log of words model.

Term-document count matrices (continued)

- Text about count matrices.

Term frequency, if

- Text about term frequency.

Term-document count matrices (continued)

- Text about count matrices.

Log frequency weighting (continued)

- Text about log frequency weighting.

Term-document count matrices (continued)

- Text about count matrices.

Document frequency, continued

- Text about document frequency.

Term-document count matrices (continued)

- Text about count matrices.

Document frequency, continued

- Text about document frequency.

Vector Space Model

Documents as vectors

- Text about documents as vectors.

Documents as vectors (continued)

- Text about documents as vectors.

Documents as vectors (continued)

- Text about documents as vectors.

Documents as vectors (continued)

- Text about documents as vectors.

Documents as vectors (continued)

- Text about documents as vectors.

Documents as vectors (continued)

- Text about documents as vectors.

Documents as vectors (continued)

- Text about documents as vectors.

Documents as vectors (continued)

- Text about documents as vectors.

Documents as vectors (continued)

- Text about documents as vectors.

Documents as vectors (continued)

- Text about documents as vectors.

Ranked retrieval

- Thus far, our queries have all been Boolean.
 - Documents either match or don't.
- Good for expert users with precise understanding of their needs and the collection.
 - Also good for applications: Applications can easily consume 1000s of results.
- Not good for the majority of users.
 - Most users incapable of writing Boolean queries (or they are, but they think it's too much work).
 - Most users don't want to wade through 1000s of results.
 - This is particularly true of web search.

Problem with Boolean search: feast or famine

- Boolean queries often result in either too few (≈ 0) or too many (1000s) results.
 - Query 1: “*standard user dlink 650*” → 200,000 hits
 - Query 2: “*standard user dlink 650 no card found*”
→ 0 hits
- It takes a lot of skill to come up with a query that produces a manageable number of hits.
 - AND gives too few; OR gives too many

Ranked retrieval models

- Rather than a set of documents satisfying a query expression, in **ranked retrieval models**, the system returns an ordering over the (top) documents in the collection with respect to a query
- **Free text queries**: Rather than a query language of operators and expressions, the user's query is just one or more words in a human language
- In principle, there are two separate choices here, but in practice, ranked retrieval models have normally been associated with free text queries and vice versa

Feast or famine: not a problem in ranked retrieval

- When a system produces a ranked result set, large result sets are not an issue
 - Indeed, the size of the result set is not an issue
 - We just show the top k (≈ 10) results
 - We don't overwhelm the user
 - Premise: the ranking algorithm works

Scoring as the basis of ranked retrieval

- We wish to return in order the documents most likely to be useful to the searcher
- How can we rank-order the documents in the collection with respect to a query?
- Assign a score – say in $[0, 1]$ – to each document
- This score measures how well document and query “match”.

Query-document matching scores

- We need a way of assigning a score to a query/document pair
- **Let's start with a one-term query**
- If the query term does not occur in the document: score should be 0
- **The more frequent the query term in the document, the higher the score (should be)**
- We will look at a number of alternatives for this

Take 1: Jaccard coefficient

- A commonly used measure of overlap of two sets A and B is the Jaccard coefficient
- $\text{jaccard}(A,B) = |A \cap B| / |A \cup B|$
- $\text{jaccard}(A,A) = 1$
- $\text{jaccard}(A,B) = 0$ if $A \cap B = 0$
- A and B don't have to be the same size.
- Always assigns a number between 0 and 1.

Jaccard coefficient: Scoring example

- What is the query-document match score that the Jaccard coefficient computes for each of the two documents below?
- Query: *ides of march*
- Document 1: *caesar died in march*
- Document 2: *the long march*

Issues with Jaccard for scoring

- It doesn't consider *term frequency* (how many times a term occurs in a document)
 - Rare terms in a collection are more informative than frequent terms
 - Jaccard doesn't consider this information
- We need a more sophisticated way of normalizing for length
 - Later in this lecture, we'll use $|A \cap B| / \sqrt{|A \cup B|}$... instead of $|A \cap B| / |A \cup B|$ (Jaccard) for length normalization.

Recall: Binary term-document incidence matrix

	Antony and Cleopatra	Julius Caesar	The Tempest	Hamlet	Othello	Macbeth
Antony	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

Each document is represented by a binary vector $\in \{0,1\}^{|V|}$

Term-document count matrices

- Consider the number of occurrences of a term in a document:
 - Each document is a count vector in $\mathbb{N}^{|V|}$: a column below

	Antony and Cleopatra	Julius Caesar	The Tempest	Hamlet	Othello	Macbeth
Antony	157	73	0	0	0	0
Brutus	4	157	0	1	0	0
Caesar	232	227	0	2	1	1
Calpurnia	0	10	0	0	0	0
Cleopatra	57	0	0	0	0	0
mercy	2	0	3	5	5	1
worser	2	0	1	1	1	0

Bag of words model

- Vector representation doesn't consider the ordering of words in a document
- *John is quicker than Mary and Mary is quicker than John have the same vectors*
- This is called the **bag of words** model.
- **In a sense, this is a step back: The positional index was able to distinguish these two documents**
 - We will look at “recovering” positional information later on
 - For now: bag of words model

Term frequency tf

- The term frequency $tf_{t,d}$ of term t in document d is defined as the number of times that t occurs in d .
- We want to use tf when computing query-document match scores. But how?
- Raw term frequency is not what we want:
 - A document with 10 occurrences of the term is more relevant than a document with 1 occurrence of the term.
 - But not 10 times more relevant.
- Relevance does not increase proportionally with term frequency.

NB: frequency = count in IR

Log-frequency weighting

- The log frequency weight of term t in d is

$$w_{t,d} = \begin{cases} 1 + \log_{10} \text{tf}_{t,d}, & \text{if } \text{tf}_{t,d} > 0 \\ 0, & \text{otherwise} \end{cases}$$

- $0 \rightarrow 0, 1 \rightarrow 1, 2 \rightarrow 1.3, 10 \rightarrow 2, 1000 \rightarrow 4$, etc.
- Score for a document-query pair: sum over terms t in both q and d :
- score
$$= \sum_{t \in q \cap d} (1 + \log \text{tf}_{t,d})$$
- The score is 0 if none of the query terms is present in the document.

Document frequency

- Rare terms are more informative than frequent terms
 - Recall stop words
- Consider a term in the query that is rare in the collection (e.g., *arachnocentric*)
- A document containing this term is very likely to be relevant to the query *arachnocentric*
- → We want a high weight for rare terms like *arachnocentric*.

Document frequency, continued

- Frequent terms are less informative than rare terms
- Consider a query term that is frequent in the collection (e.g., *high*, *increase*, *line*)
- A document containing such a term is more likely to be relevant than a document that doesn't
- But it's not a sure indicator of relevance.
- → For frequent terms, we want positive weights for words like *high*, *increase*, and *line*
- But lower weights than for rare terms.
- We will use document frequency (df) to capture this.

idf weight

- df_t is the document frequency of t : the number of documents that contain t
 - df_t is an inverse measure of the informativeness of t
 - $df_t \leq N$
- We define the idf (inverse document frequency) of t by $idf_t = \log_{10} (N/df_t)$

- We use $\log (N/df_t)$ instead of N/df_t to “dampen” the effect of idf.

Will turn out the base of the log is immaterial.

idf example, suppose $N = 1$ million

term	df_t	idf_t
calpurnia	1	
animal	100	
sunday	1,000	
fly	10,000	
under	100,000	
the	1,000,000	

$$idf_t = \log_{10} (N/df_t)$$

There is one idf value for each term t in a collection.

Effect of idf on ranking

- Question: Does idf have an effect on ranking for one-term queries, like
 - iPhone
- idf has no effect on ranking one term queries
 - idf affects the ranking of documents for queries with at least two terms
 - For the query **capricious person**, idf weighting makes occurrences of **capricious** count for much more in the final document ranking than occurrences of **person**.

Collection vs. Document frequency

- The collection frequency of t is the number of occurrences of t in the collection, counting multiple occurrences.
- Example:

Word	Collection frequency	Document frequency
<i>insurance</i>	10440	3997
<i>try</i>	10422	8760

- Which word is a better search term (and should get a higher weight)?

tf-idf weighting

- The tf-idf weight of a term is the product of its tf weight and its idf weight.

$$w_{t,d} = (1 + \log \text{tf}_{t,d}) \times \log_{10}(N / \text{df}_t)$$

- **Best known weighting scheme in information retrieval**
 - Note: the “-” in tf-idf is a hyphen, not a minus sign!
 - **Alternative names: tf.idf, tf x idf**
- Increases with the number of occurrences within a document
- **Increases with the rarity of the term in the collection**

Final ranking of documents for a query

$$\text{Score}(q, d) = \sum_{t \in q \cap d} \text{tf.idf}_{t, d}$$

Binary \rightarrow count \rightarrow weight matrix

	Antony and Cleopatra	Julius Caesar	The Tempest	Hamlet	Othello	Macbeth
Antony	5.25	3.18	0	0	0	0.35
Brutus	1.21	6.1	0	1	0	0
Caesar	8.59	2.54	0	1.51	0.25	0
Calpurnia	0	1.54	0	0	0	0
Cleopatra	2.85	0	0	0	0	0
mercy	1.51	0	1.9	0.12	5.25	0.88
worser	1.37	0	0.11	4.15	0.25	1.95

Each document is now represented by a real-valued vector of tf-idf weights $\in \mathbb{R}^{|V|}$

Ranked Retrieval

Ranked retrieval

- It takes as input a query and returns a ranked list of documents
- It is a special case of information retrieval
- It is a special case of information retrieval
- It is a special case of information retrieval
- It is a special case of information retrieval

Term-document count matrices

- Consider the number of occurrences of a term in a document
- Consider the number of occurrences of a term in a document
- Consider the number of occurrences of a term in a document
- Consider the number of occurrences of a term in a document

...

Problem with naive search

- Problem with naive search: it often looks for the best match
- Problem with naive search: it often looks for the best match
- Problem with naive search: it often looks for the best match

BoW of words model

- The naive search algorithm is often based on the BoW model
- The naive search algorithm is often based on the BoW model
- The naive search algorithm is often based on the BoW model

Ranked retrieval models

- There are several ranked retrieval models
- There are several ranked retrieval models
- There are several ranked retrieval models

Term frequency/df

- Term frequency (TF) is the number of times a term appears in a document
- Term frequency (TF) is the number of times a term appears in a document
- Term frequency (TF) is the number of times a term appears in a document

Fast or familiar: not a problem in ranked retrieval

- Fast or familiar: not a problem in ranked retrieval
- Fast or familiar: not a problem in ranked retrieval
- Fast or familiar: not a problem in ranked retrieval

Log-frequency weighting

- Log-frequency weighting is used to reduce the impact of common terms
- Log-frequency weighting is used to reduce the impact of common terms
- Log-frequency weighting is used to reduce the impact of common terms

Scoring on the basis of ranked retrieval

- Scoring on the basis of ranked retrieval
- Scoring on the basis of ranked retrieval
- Scoring on the basis of ranked retrieval

Document frequency

- Document frequency (DF) is the number of documents containing a term
- Document frequency (DF) is the number of documents containing a term
- Document frequency (DF) is the number of documents containing a term

Document frequency, continued

- Document frequency, continued
- Document frequency, continued
- Document frequency, continued

df weight

- df weight is used to weight terms based on their document frequency
- df weight is used to weight terms based on their document frequency
- df weight is used to weight terms based on their document frequency

Jaccard coefficient: starting example

- Jaccard coefficient: starting example
- Jaccard coefficient: starting example
- Jaccard coefficient: starting example

idf estimate, suppose N = 1 million

- idf estimate, suppose N = 1 million
- idf estimate, suppose N = 1 million
- idf estimate, suppose N = 1 million

Issues with Jaccard for scoring

- Issues with Jaccard for scoring
- Issues with Jaccard for scoring
- Issues with Jaccard for scoring

Effect of idf on ranking

- Effect of idf on ranking
- Effect of idf on ranking
- Effect of idf on ranking

Recall: Binary term-document incidence matrix

Document	Term 1	Term 2	Term 3	Term 4
D1	1	1	0	1
D2	1	0	1	1
D3	0	1	1	0
D4	1	1	1	1

...

Collection vs. Document frequency

- Collection vs. Document frequency
- Collection vs. Document frequency
- Collection vs. Document frequency

Vector Space Model

Embedding vectors

- Embedding vectors
- Embedding vectors
- Embedding vectors

Geometrically, B, Matrix

- Geometrically, B, Matrix
- Geometrically, B, Matrix
- Geometrically, B, Matrix

Embedding vectors

- Embedding vectors
- Embedding vectors
- Embedding vectors

Embedding vectors

- Embedding vectors
- Embedding vectors
- Embedding vectors

Embedding vectors

- Embedding vectors
- Embedding vectors
- Embedding vectors

Embedding vectors

- Embedding vectors
- Embedding vectors
- Embedding vectors

df weight

- df weight
- df weight
- df weight

df weight

- df weight
- df weight
- df weight

Binary → count → weight matrix

- Binary → count → weight matrix
- Binary → count → weight matrix
- Binary → count → weight matrix

Binary → count → weight matrix

- Binary → count → weight matrix
- Binary → count → weight matrix
- Binary → count → weight matrix

Binary → count → weight matrix

- Binary → count → weight matrix
- Binary → count → weight matrix
- Binary → count → weight matrix

Binary → count → weight matrix

- Binary → count → weight matrix
- Binary → count → weight matrix
- Binary → count → weight matrix

Binary → count → weight matrix

- Binary → count → weight matrix
- Binary → count → weight matrix
- Binary → count → weight matrix

Binary → count → weight matrix

- Binary → count → weight matrix
- Binary → count → weight matrix
- Binary → count → weight matrix

Binary → count → weight matrix

- Binary → count → weight matrix
- Binary → count → weight matrix
- Binary → count → weight matrix

Binary → count → weight matrix

- Binary → count → weight matrix
- Binary → count → weight matrix
- Binary → count → weight matrix

Binary → count → weight matrix

- Binary → count → weight matrix
- Binary → count → weight matrix
- Binary → count → weight matrix

Binary → count → weight matrix

- Binary → count → weight matrix
- Binary → count → weight matrix
- Binary → count → weight matrix

Binary → count → weight matrix

- Binary → count → weight matrix
- Binary → count → weight matrix
- Binary → count → weight matrix

Binary → count → weight matrix

- Binary → count → weight matrix
- Binary → count → weight matrix
- Binary → count → weight matrix

Documents as vectors

- Now we have a $|V|$ -dimensional vector space
- **Terms are axes of the space**
- Documents are points or vectors in this space
- **Very high-dimensional: tens of millions of dimensions when you apply this to a web search engine**
- These are very sparse vectors – most entries are zero

Queries as vectors

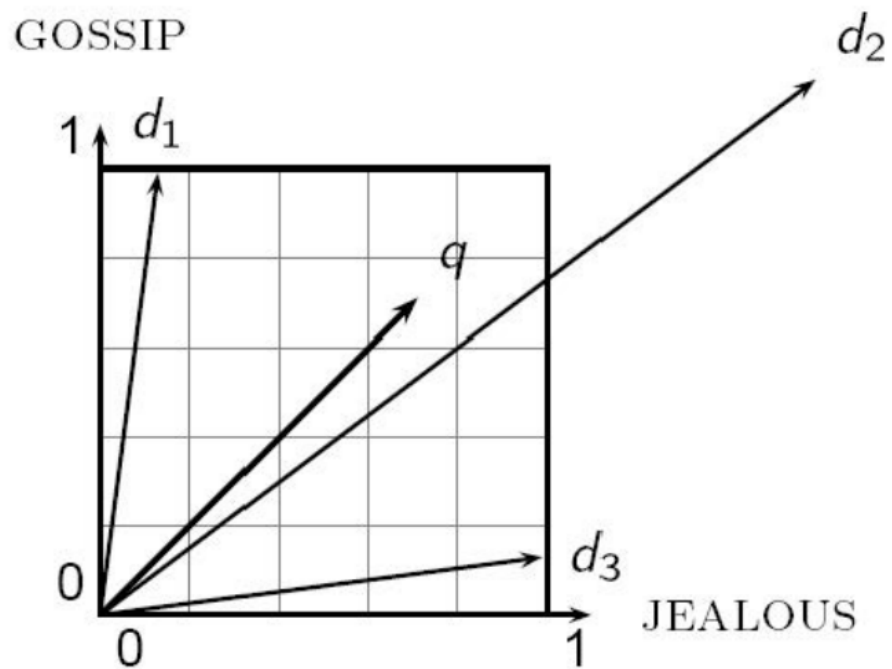
- Key idea 1: Do the same for queries: represent them as vectors in the space
- Key idea 2: Rank documents according to their proximity to the query in this space
- proximity = similarity of vectors
- proximity \approx inverse of distance
- **Recall: We do this because we want to get away from the you're-either-in-or-out Boolean model**
- Instead: rank more relevant documents higher than less relevant documents

Formalizing vector space proximity

- First cut: distance between two points
 - (= distance between the end points of the two vectors)
- **Euclidean distance?**
- Euclidean distance is a bad idea . . .
- . . . because Euclidean distance is **large** for vectors of **different lengths**.

Why distance is a bad idea

The Euclidean distance between q and d_2 is large even though the distribution of terms in the query q and the distribution of terms in the document d_2 are very similar.



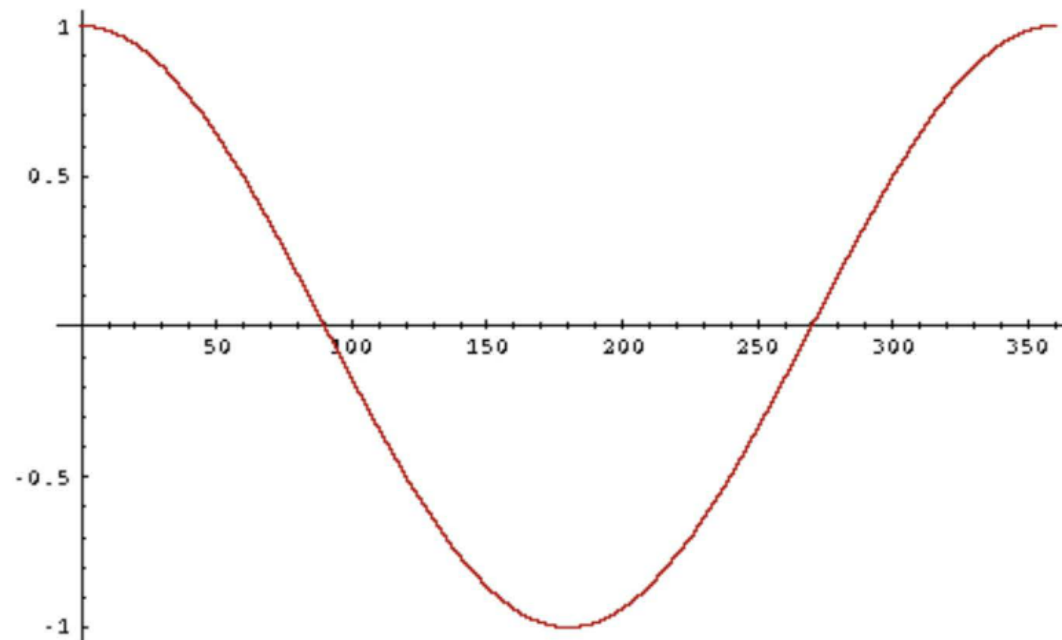
Use angle instead of distance

- Thought experiment: take a document d and append it to itself. Call this document d' .
- “Semantically” d and d' have the same content
- The Euclidean distance between the two documents can be quite large
- The angle between the two documents is 0, corresponding to maximal similarity.
- Key idea: Rank documents according to angle with query.

From angles to cosines

- The following two notions are equivalent.
 - Rank documents in decreasing order of the angle between query and document
 - Rank documents in increasing order of $\text{cosine}(\text{query}, \text{document})$
- Cosine is a monotonically decreasing function for the interval $[0^\circ, 180^\circ]$

From angles to cosines



- But how – *and why* – should we be computing cosines?

Length normalization

- A vector can be (length-) normalized by dividing each of its components by its length – for this we use the L_2 norm:

$$\|\vec{x}\|_2 = \sqrt{\sum_i x_i^2}$$

- Dividing a vector by its L_2 norm makes it a unit (length) vector (on surface of unit hypersphere)
- Effect on the two documents d and d' (d appended to itself) from earlier slide: they have identical vectors after length-normalization.
 - Long and short documents now have comparable weights

cosine(query, document)

$$\cos(\vec{q}, \vec{d}) = \frac{\vec{q} \cdot \vec{d}}{|\vec{q}| |\vec{d}|} = \frac{\vec{q}}{|\vec{q}|} \cdot \frac{\vec{d}}{|\vec{d}|} = \frac{\sum_{i=1}^{|\mathcal{V}|} q_i d_i}{\sqrt{\sum_{i=1}^{|\mathcal{V}|} q_i^2} \sqrt{\sum_{i=1}^{|\mathcal{V}|} d_i^2}}$$

q_i is the tf-idf weight of term i in the query

d_i is the tf-idf weight of term i in the document

$\cos(\vec{q}, \vec{d})$ is the cosine similarity of \vec{q} and \vec{d} ... or, equivalently, the cosine of the angle between \vec{q} and \vec{d} .

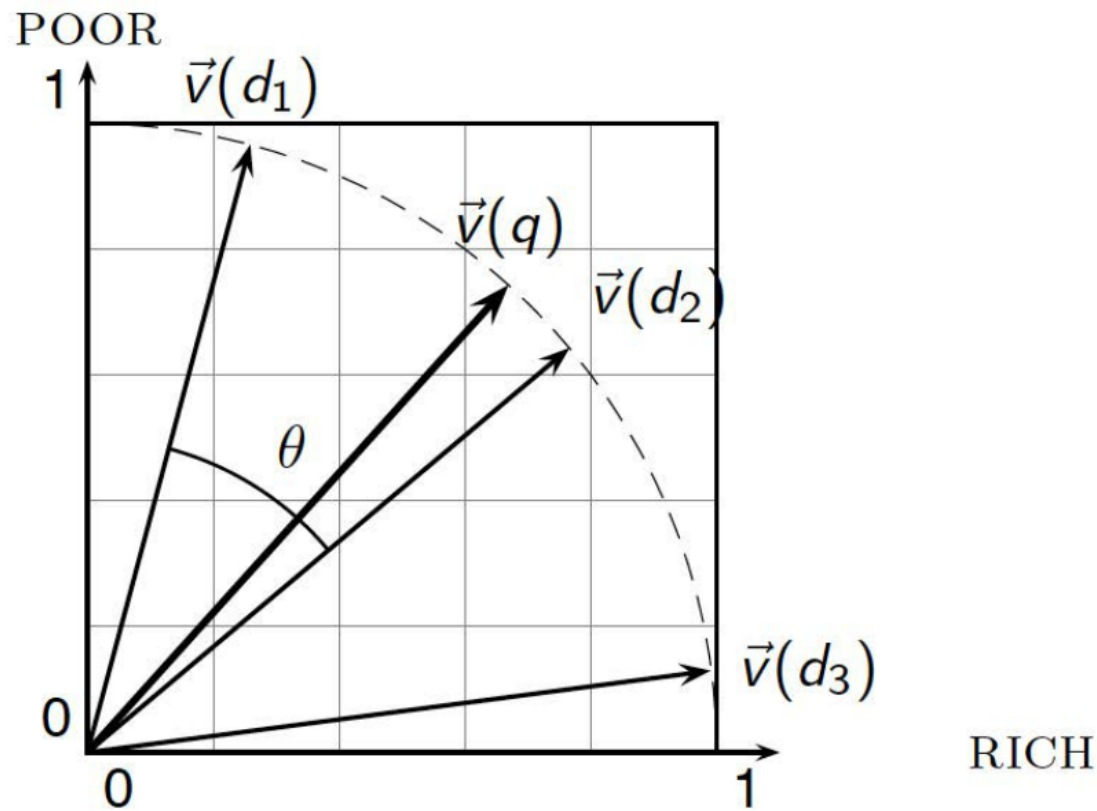
Cosine for length-normalized vectors

- For length-normalized vectors, cosine similarity is simply the dot product (or scalar product):

$$\cos(\vec{q}, \vec{d}) = \vec{q} \cdot \vec{d} = \sum_{i=1}^{|\mathcal{V}|} q_i d_i$$

for q, d length-normalized.

Cosine similarity illustrated



Cosine similarity amongst 3 documents

How similar are
the novels

SaS: *Sense and
Sensibility*

PaP: *Pride and
Prejudice*, and

WH: *Wuthering
Heights*?

term	SaS	PaP	WH
affection	115	58	20
jealous	10	7	11
gossip	2	0	6
wuthering	0	0	38

Term frequencies (counts)

Note: To simplify this example, we don't do idf weighting.

3 documents example contd.

Log frequency weighting

term	SaS	PaP	WH
affection	3.06	2.76	2.30
jealous	2.00	1.85	2.04
gossip	1.30	0	1.78
wuthering	0	0	2.58

After length normalization

term	SaS	PaP	WH
affection	0.789	0.832	0.524
jealous	0.515	0.555	0.465
gossip	0.335	0	0.405
wuthering	0	0	0.588

$\cos(\text{SaS}, \text{PaP}) \approx$

$$0.789 \times 0.832 + 0.515 \times 0.555 + 0.335 \times 0.0 + 0.0 \times 0.0 \approx \mathbf{0.94}$$

$\cos(\text{SaS}, \text{WH}) \approx \mathbf{0.79}$

$\cos(\text{PaP}, \text{WH}) \approx \mathbf{0.69}$

Why do we have $\cos(\text{SaS}, \text{PaP}) > \cos(\text{SAS}, \text{WH})$?

