

Text Mining

Laura Brown

Some slides adapted from P. Smyth; Han, Kamber, & Pei;
Tan, Steinbach, & Kumar; C. Volinsky; R. Tibshirani; D. Kauchak
and <http://nlp.stanford.edu/IR-book/>

Outline

- Information Retrieval
 - What is it?
 - Challenges
 - Boolean Queries
 - Phrase Queries
 - Ranked Retrieval

Information retrieval (IR)

- What comes to mind when I say “information retrieval”?
- Where have you seen IR? What are some real-world examples/uses?
 - Search engines
 - File search (e.g. OS X Spotlight, Windows Instant Search, Google Desktop)
 - Databases?
 - Catalog search (e.g. library)
 - Intranet search (i.e. corporate networks)

Information Retrieval

- Information Retrieval is finding material in text documents of an unstructured nature that satisfy an information need from within large collections of digitally stored content

Information Retrieval

- Information Retrieval is finding material in text documents of an unstructured nature that satisfy an information need from within large collections of digitally stored content
 - Find all documents about computer science
 - Find all course web pages at Michigan Tech
 - What is the cheapest flight from LA to NY?
 - Who is was the 15th president?

Information Retrieval

- Information Retrieval is finding material in text documents of an unstructured nature that satisfy an information need from within large collections of digitally stored content

What is the difference between an *information need* and a *query*?

Information Retrieval

- Information Retrieval is finding material in text documents of an unstructured nature that satisfy an information need from within large collections of digitally stored content

Information need

- Find all documents about computer science
- Find all course web pages at Michigan Tech
- Who is was the 15th president?

Query

“computer science”

Michigan Tech AND college
AND *url-contains* class

WHO=president NUMBER=15

IR vs. databases

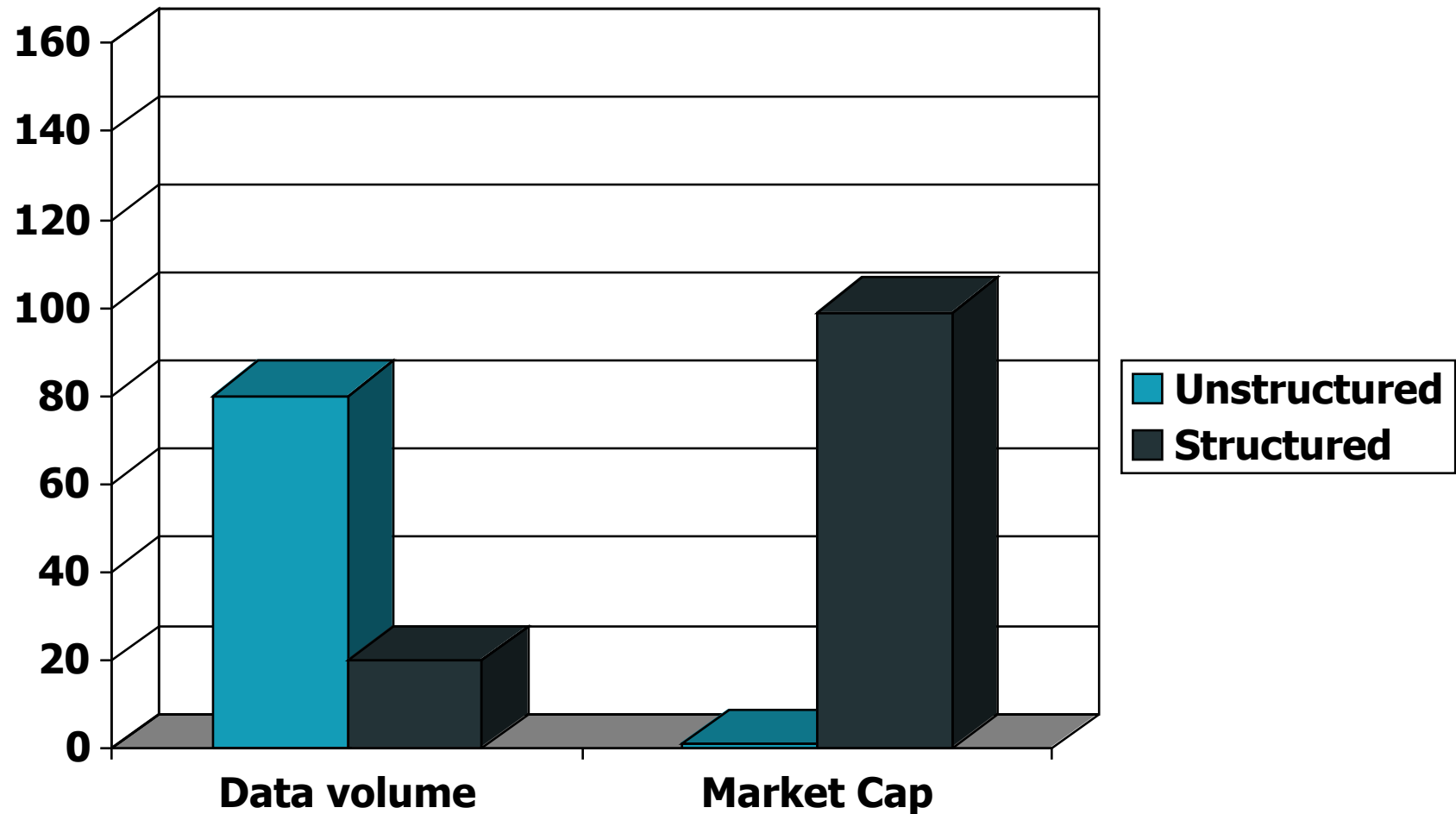
- Structured data tends to refer to information in “tables”

Employee	Manager	Salary
Smith	Jones	50000
Chang	Smith	60000
Ivy	Smith	50000

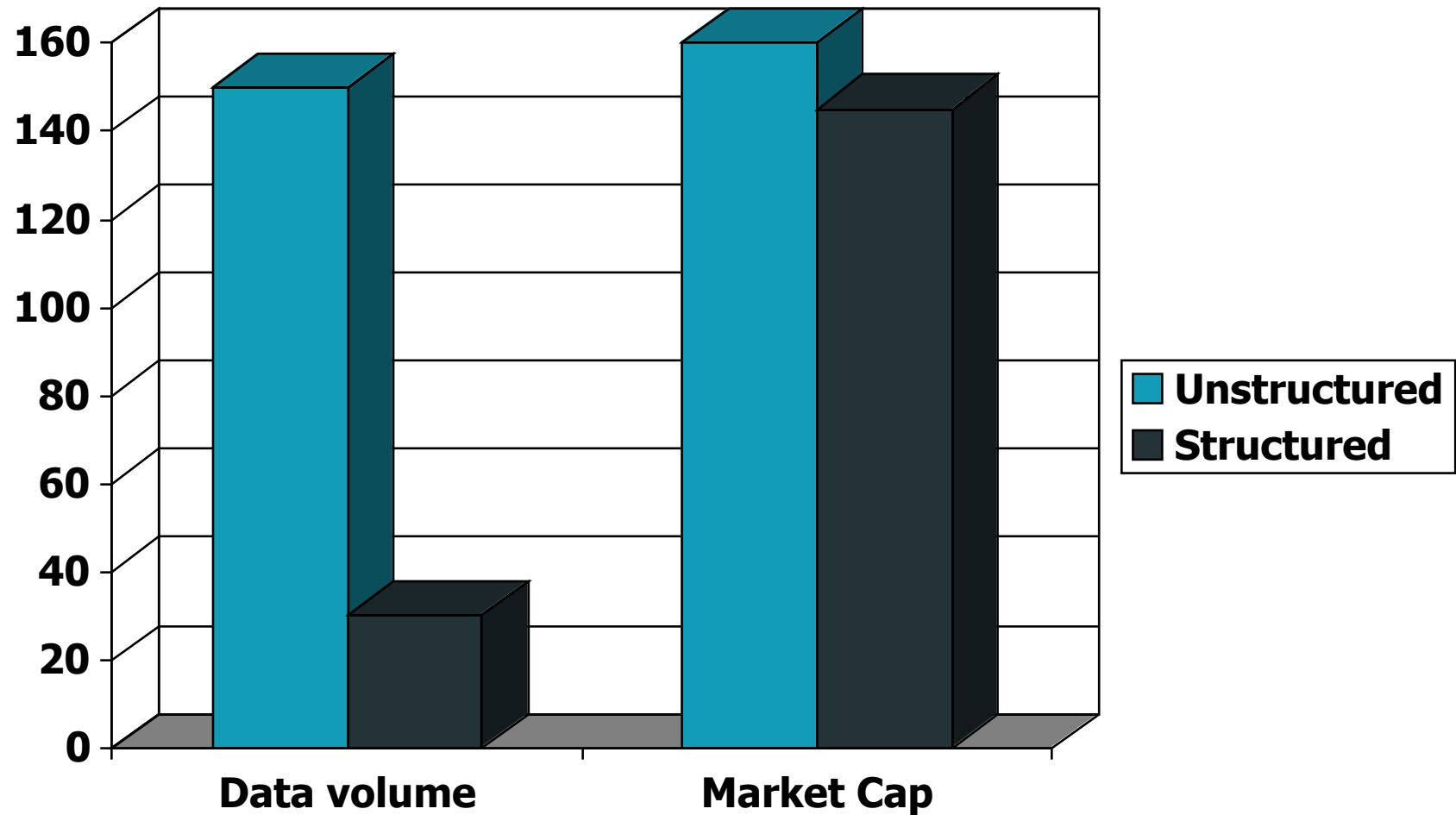
Typically allows numerical range and exact match (for text) queries, e.g.,

Salary < 60000 AND Manager = Smith.

Unstructured (text) vs. structured (database) data in 1996



Unstructured (text) vs. structured (database) data in 2006



Information Retrieval

- Find k objects in the corpus of documents which are most similar to my query
- Can be viewed as “interactive” data mining
 - query not specified a priori
- Main problems with text retrieval:
 - What does similar mean?
 - How do you know if you have the right documents?
 - How can user feedback be incorporated?

Information Retrieval: Challenges

- Calculating similarity is not obvious - what is the distance between two sentences or queries?
- Evaluating retrieval is hard: what is the “right” answer ? (no ground truth)
- User can query things you have not seen before e.g. misspelled, foreign, new terms.
- Goal (score function) is different than in classification/regression: not looking to model all of the data, just get best results for a given user.
- Words can hide semantic content
 - **Synonymy**: A keyword T does not appear anywhere in the document, even though the document is closely related to T , e.g., data mining
 - **Polysemy**: The same keyword may mean different things in different contexts, e.g., mining

Information Retrieval: Practices

- Representation language
 - typically a vector of d attributes, e.g.,
 - set of color, intensity, texture, features of images
 - word counts for text documents
- Data set D of N objects
 - represented as a $N \times d$ matrix
- Query Q
 - user poses query to search D
 - query is typically expressed in same representation language as the data, e.g.,
 - each text document is a set of words that occur in the document

Query by Content

- traditional DB query: exact matches
 - e.g. query $Q = [\text{level} = \text{MANAGER}] \text{ AND } [\text{age} < 30]$
 - or, Boolean match on text
 - query = “Irvine” AND “fun”: return all docs with “Irvine” and “fun”
 - Not useful when there are many matches
 - E.g., “data mining” in Google returns 60 million documents

Query by Content

- query-by-content query: more general / less precise
 - e.g. what record is most similar to a query Q?
 - for text data, often called “information retrieval (IR)”
 - can also be used for images, sequences, video, etc
 - Q can itself be an object (e.g., a document) or a shorter version (e.g., 1 word)
- Goal
 - Match query Q to the N objects in the database
 - Return a ranked list (typically) of the most similar/relevant objects in the data set D given Q

Example: Unstructured data in 1680

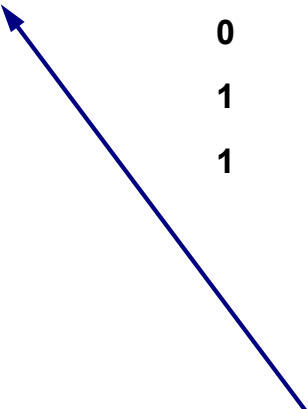
- Which plays of Shakespeare contain the words ***Brutus AND Caesar*** but ***NOT Calpurnia***?
- One could grep all of Shakespeare's plays for ***Brutus*** and ***Caesar***, then strip out plays containing ***Calpurnia***. Any problems with this?
 - Slow (for large corpora)
 - Other operations (e.g., find the word ***Romans*** near ***countrymen***) not feasible
 - Ranked retrieval (best documents to return)
 - Later lectures

Example: Unstructured data in 1680

- Which plays of Shakespeare contain the words ***Brutus AND Caesar*** but ***NOT Calpurnia***?
- How might we speed up this type of query?
- Indexing: for each word, keep track of which documents it occurs in

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



1 if play contains
word, 0 otherwise

Incidence vectors

- For each term, we have a 0/1 vector
 - Caesar = 110111
 - Brutus = 110100
 - Calpurnia = 010000

	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

Incidence vectors

- For each term, we have a 0/1 vector
 - Caesar = 110111
 - Brutus = 110100
 - Calpurnia = 010000

How can we get the answer from these vectors?

Incidence vectors

- For each term, we have a 0/1 vector
 - Caesar = 110111
 - Brutus = 110100
 - Calpurnia = 010000
- Bitwise AND the vectors together using the complemented vector for all NOT queries
- Caesar AND Brutus AND COMPLEMENT(Calpurnia)
 - $110111 \& 110100 \& \sim 010000 =$
 - $110111 \& 110100 \& 101111 =$
 - 100100

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.



Incidence vectors

- For each term, we have a 0/1 vector
 - Caesar = 110111
 - Brutus = 110100
 - Calpurnia = 010000
- Bitwise AND the vectors together using the complemented vector for all NOT queries

Any problem with this approach?

Bigger collections

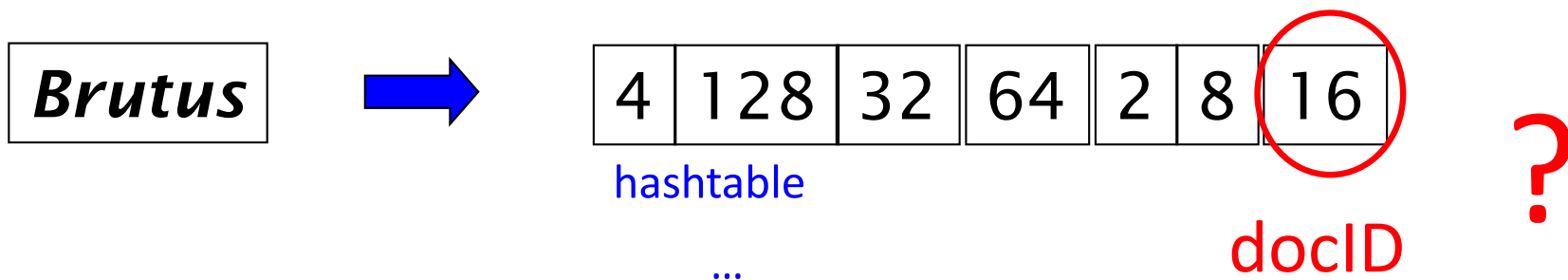
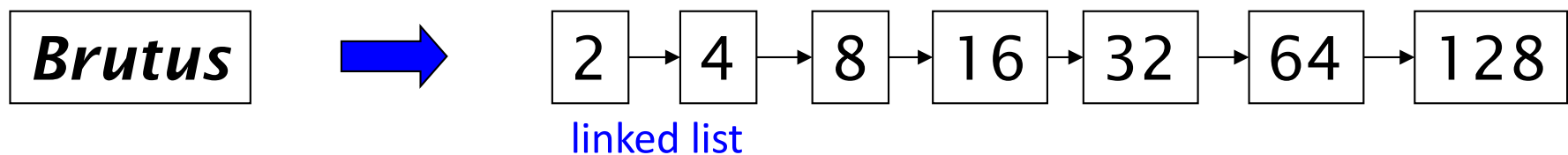
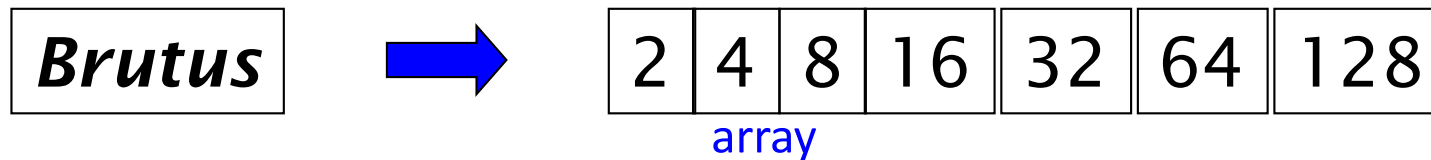
- Consider $N = 1$ million documents, each with about 1000 words
- Say there are $M = 500\text{K}$ distinct terms among these. How big is the incidence matrix?
- The matrix is a 500K by 1 million matrix = half a trillion 0's and 1's
 - Even for a moderate sized data set we can't store the matrix in memory
- Each vector has 1 million entries
 - Bitwise operations become much more expensive

What does the matrix look like?

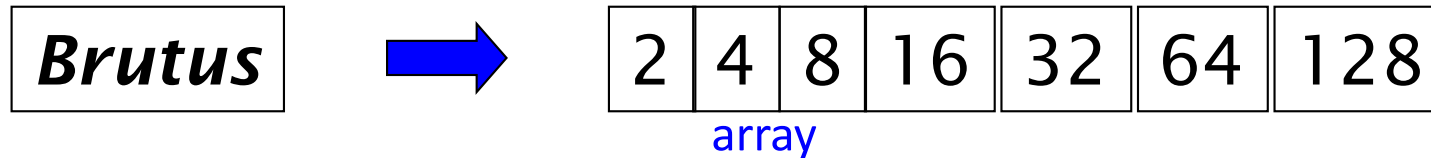
- Consider $N = 1$ million documents, each with about 1000 words
- Extremely sparse!
- How many 1's does the matrix contain?
 - no more than one billion
 - Each of the 1 million documents has at most 1000 1's
 - In practice, we'll see that the number of unique words in a document is much less than this
- What's a better representation?
 - Only record the 1 positions

Inverted index

- For each term, we store a list of all documents that contain it
- What data structures might we use for this?

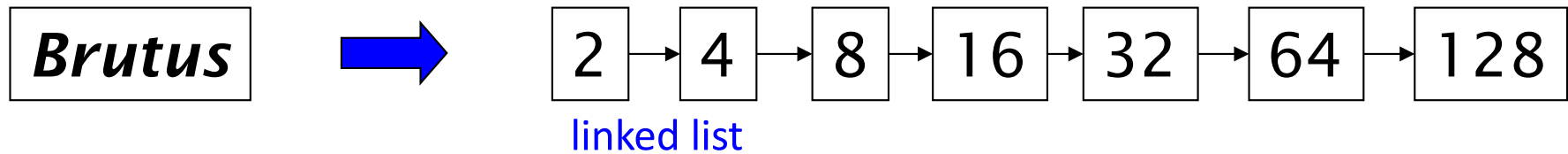


Inverted index representation



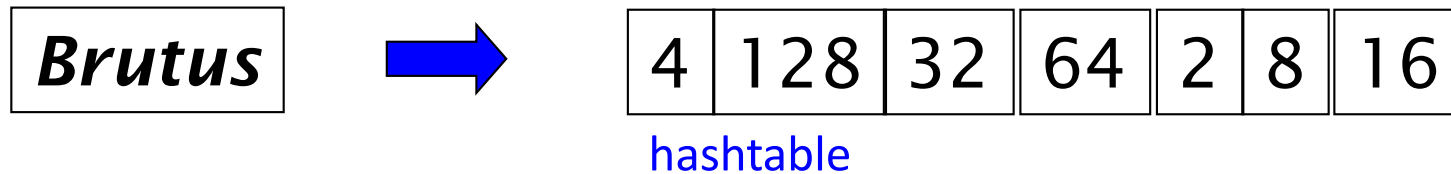
- Pros
 - Simple to implement
 - No extra pointers required for data structure
 - Contiguous memory
- Cons
 - How do we pick the size of the array?
 - What if we want to add additional documents?

Inverted index representation



- Pros
 - Dynamic space allocation
 - Insertion of new documents is straightforward
- Cons
 - Memory overhead of pointers
 - Noncontiguous memory access

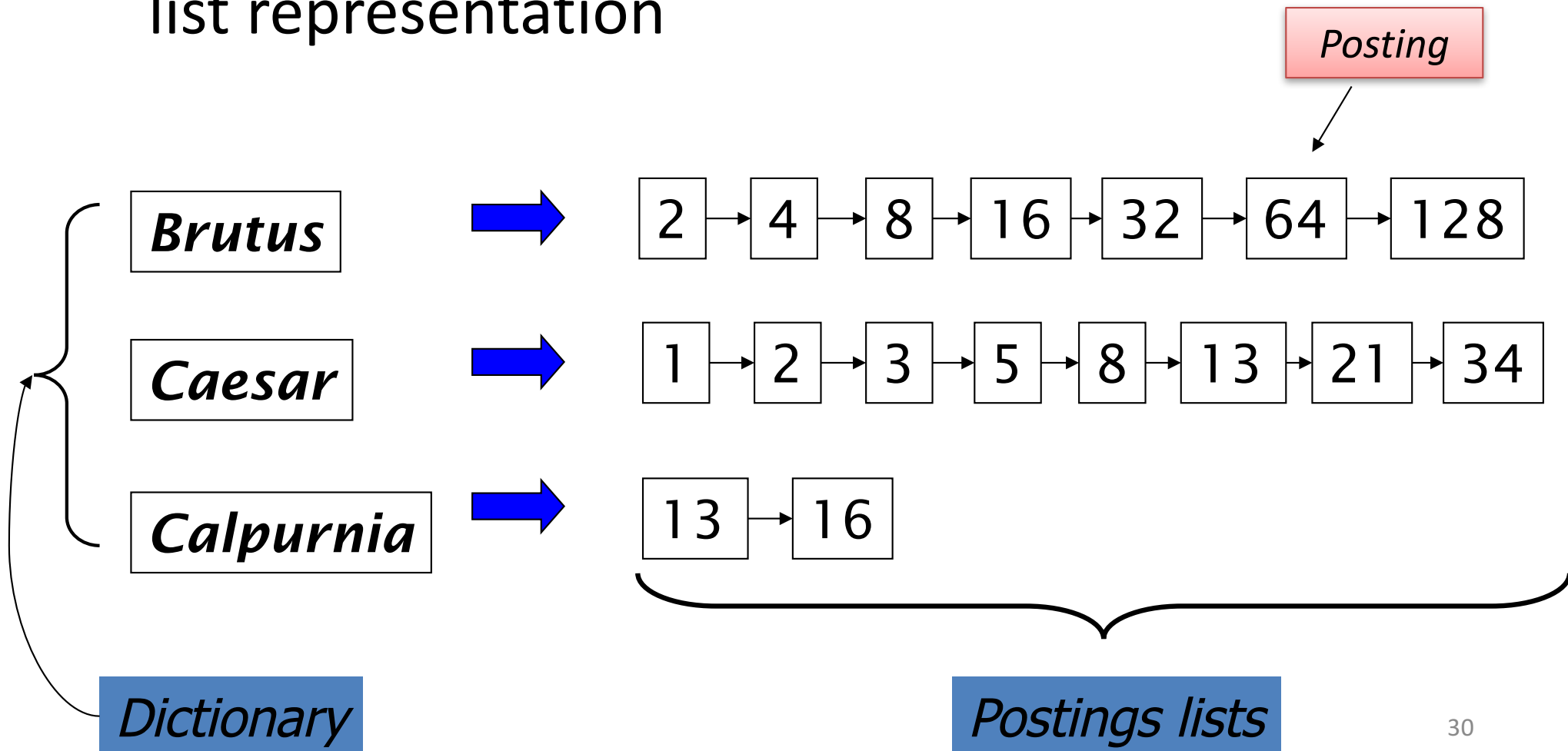
Inverted index representation



- Pros
 - Search in constant time
 - Contiguous memory
- Cons
 - How do we pick the size?
 - What if we want to add additional documents?
 - May have to rehash if we increase in size
 - To get constant time operations, lots of unused slots/memory

Inverted index

- The most common approach is to use a linked list representation



Inverted index construction

Documents to
be indexed



Friends, Romans, countrymen.

text preprocessing

friend , roman , countrymen .

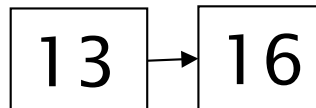
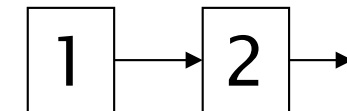
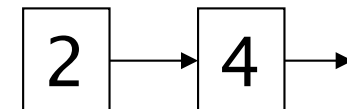
indexer

Inverted index

friend

roman

countryman



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
	33

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
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
	34

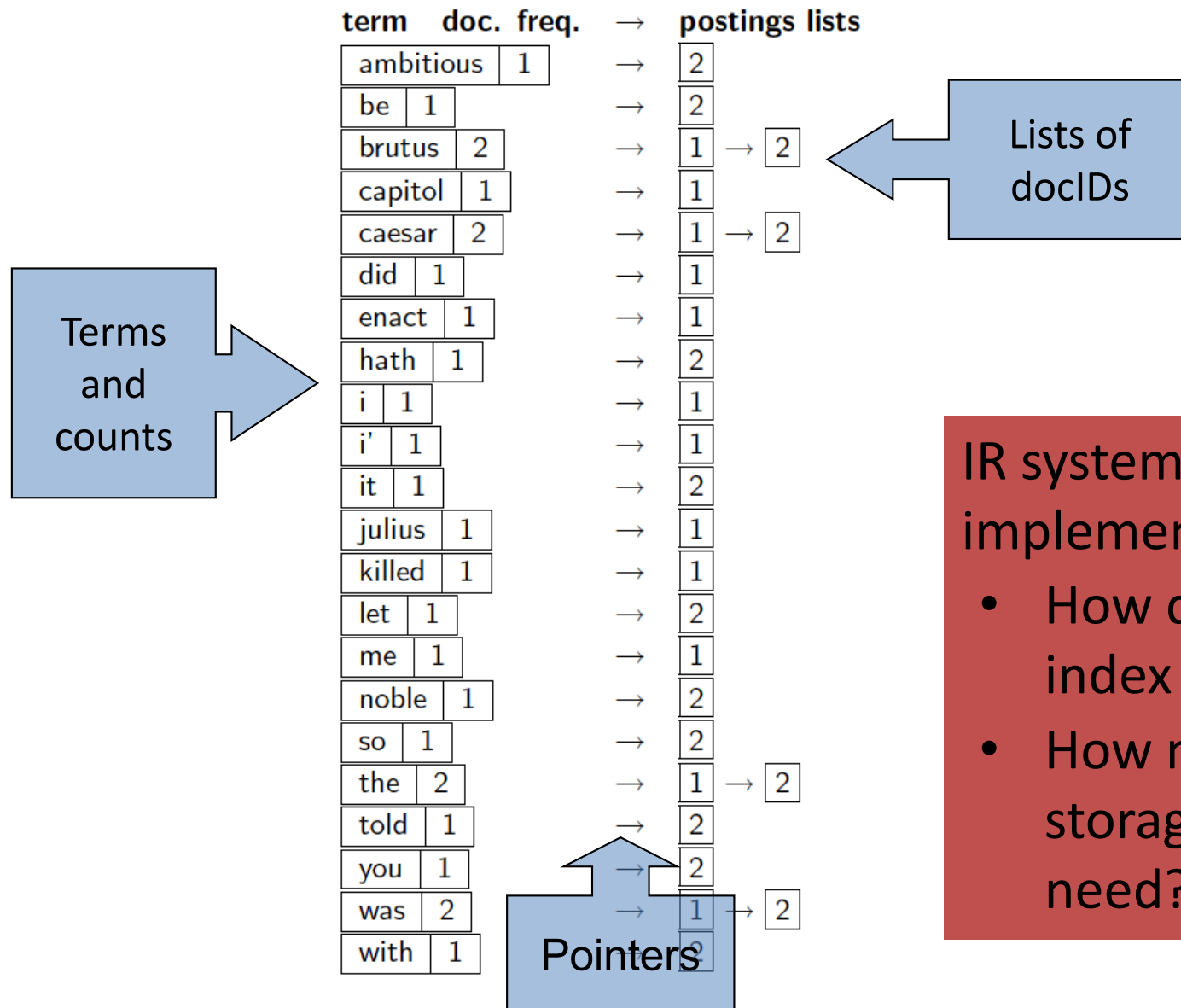
Indexer steps: Dictionary & Postings

- Multiple term entries in a single document are merged.
- Split into Dictionary and Postings
- Doc. frequency information is added.

Why frequency?
Will discuss later.

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

Where do we pay in storage?



IR system implementation

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

Boolean retrieval

- In the boolean retrieval model we ask a query that is a boolean expression:
 - A boolean query uses *AND*, *OR* and *NOT* to join query terms
 - Caesar *AND* Brutus *AND NOT* Calpurnia
 - Michigan *AND* Tech
 - (Mike *OR* Michael) *AND* Jordan *AND NOT* (Nike *OR* Gatorade)
- Given only these operations, what types of questions **can't** we answer?
 - Phrases, e.g. “Michigan Tech”
 - Proximity, “Michael” within 2 words of “Jordan”

Boolean retrieval

- Primary commercial retrieval tool for 3 decades
- Professional searchers (e.g., lawyers) still like boolean queries
- **Why?**
 - You know exactly what you're getting, a query either matches or it doesn't
 - Through trial and error, can frequently fine tune the query appropriately
 - Don't have to worry about underlying heuristics (e.g. PageRank, term weightings, synonym, etc...)

Example: WestLaw <http://www.westlaw.com/>

- Largest commercial (paying subscribers) legal search service (started 1975; ranking added 1992)
- Tens of terabytes of data; 700,000 users
- Majority of users *still* use boolean queries
- Example query:
 - What is the statute of limitations in cases involving the federal tort claims act?
 - **LIMIT /3 STATUTE ACTION /S FEDERAL /2 TORT /3 CLAIM**
 - All words starting with “LIMIT”

Example: WestLaw <http://www.westlaw.com/>

- Largest commercial (paying subscribers) legal search service (started 1975; ranking added 1992)
- Tens of terabytes of data; 700,000 users
- Majority of users *still* use boolean queries
- Example query:
 - What is the statute of limitations in cases involving the federal tort claims act?
 - LIMIT! /3 STATUTE ACTION /S FEDERAL /2 TORT /3 CLAIM

Example: WestLaw <http://www.westlaw.com/>

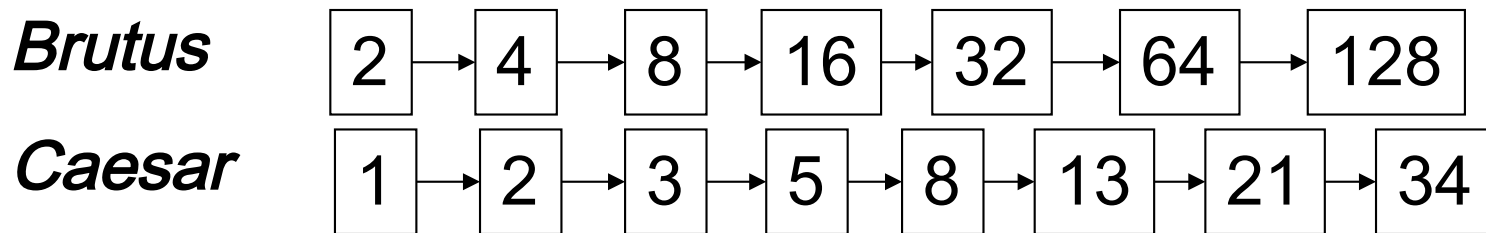
- Largest commercial (paying subscribers) legal search service (started 1975; ranking added 1992)
- Tens of terabytes of data; 700,000 users
- Majority of users *still* use boolean queries
- Example query:
 - What is the statute of limitations in cases involving the federal tort claims act?
 - **LIMIT! /3 STATUTE ACTION /S FEDERAL /2 TORT /3 CLAIM**
 - /3 = within 3 words, /S = in same sentence

Example: WestLaw <http://www.westlaw.com/>

- Another example query:
 - Requirements for disabled people to be able to access a workplace
 - `disabl! /p acces\s! /s work-site work-place (employment /3 place)`
- Long, precise queries; proximity operators; incrementally developed; not like web search
- Professional searchers often like Boolean search:
 - Precision, transparency and control
- But that doesn't mean they actually work better....

Query processing: AND

- What needs to happen to process:
Brutus* AND *Caesar
- Locate ***Brutus*** and ***Caesar*** in the Dictionary;
 - Retrieve postings lists

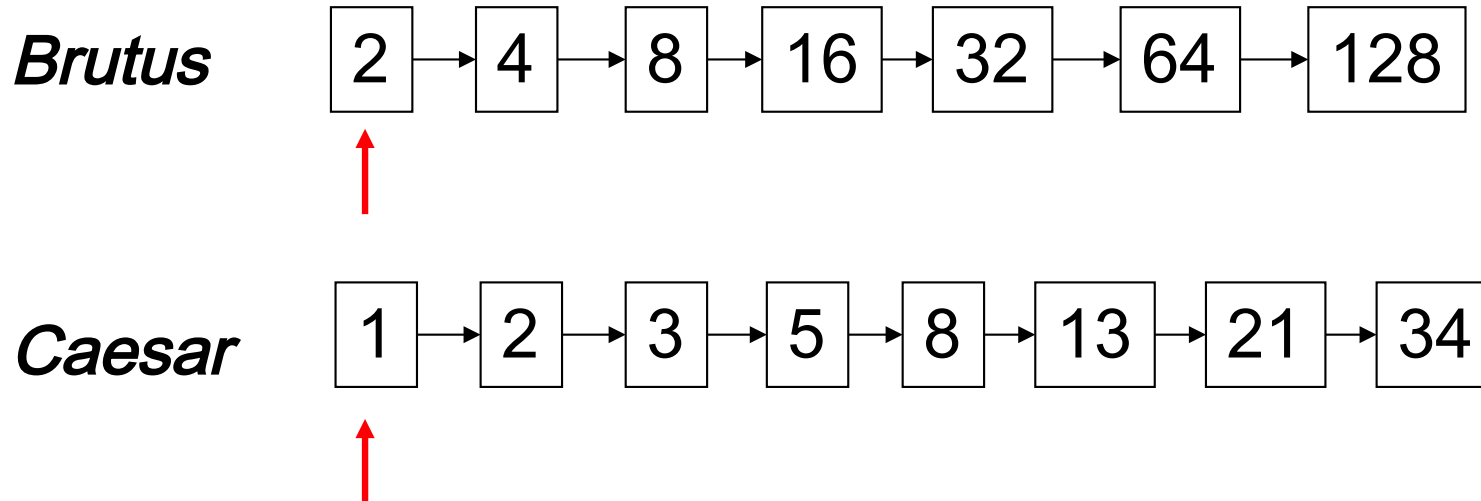


- “Merge” the two postings:



The merge

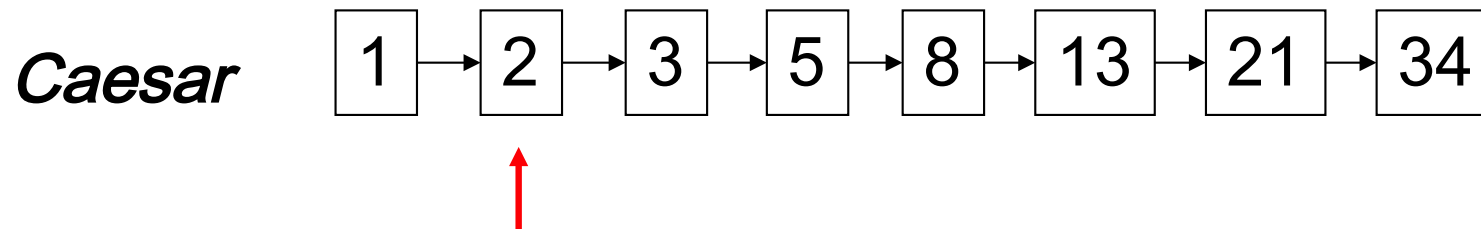
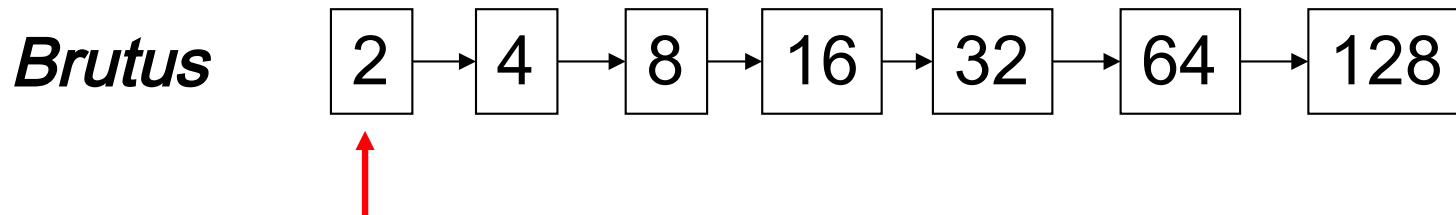
- Walk through the two postings simultaneously



Brutus AND Caesar

The merge

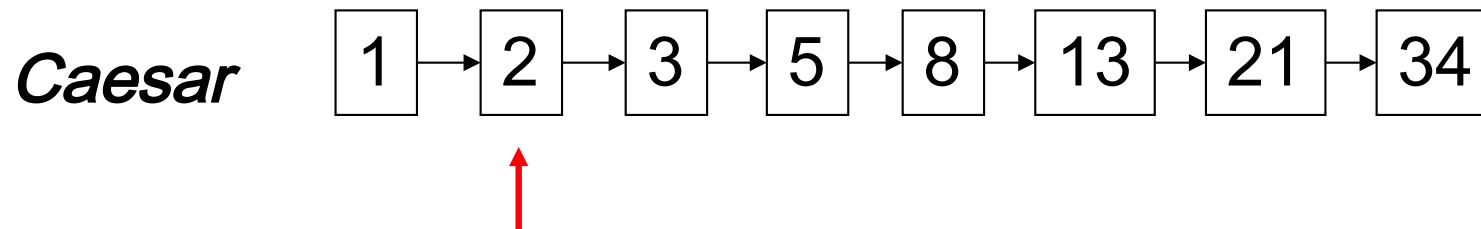
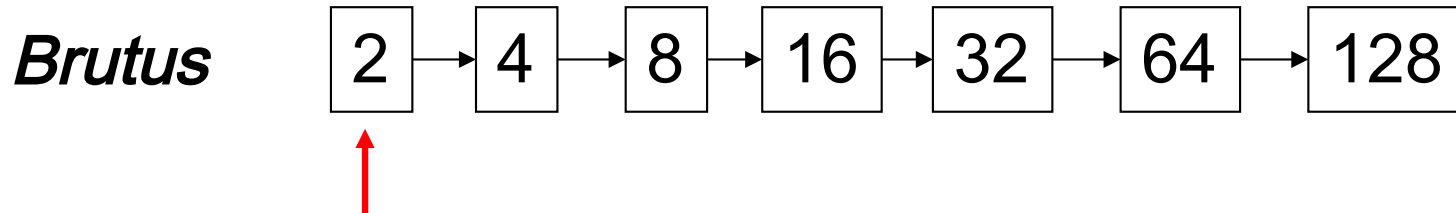
- Walk through the two postings simultaneously



Brutus AND Caesar

The merge

- Walk through the two postings simultaneously



The merge

- Walk through the two postings simultaneously

Brutus

2

 →

4

 →

8

 →

16

 →

32

 →

64

 →

128



Caesar

1

 →

2

 →

3

 →

5

 →

8

 →

13

 →

21

 →

34

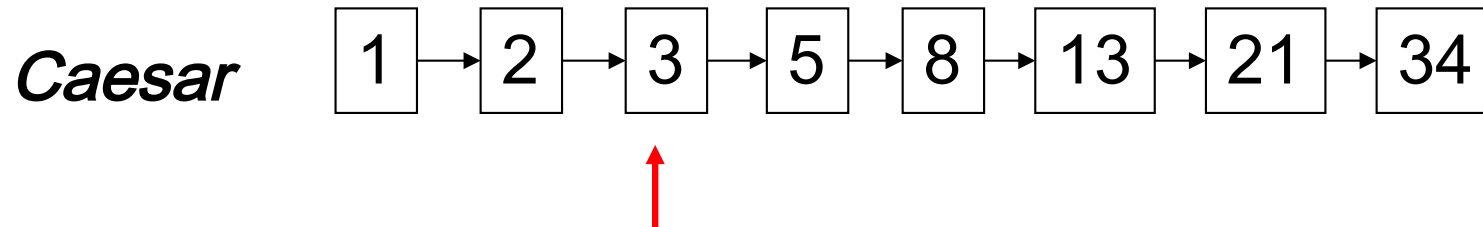
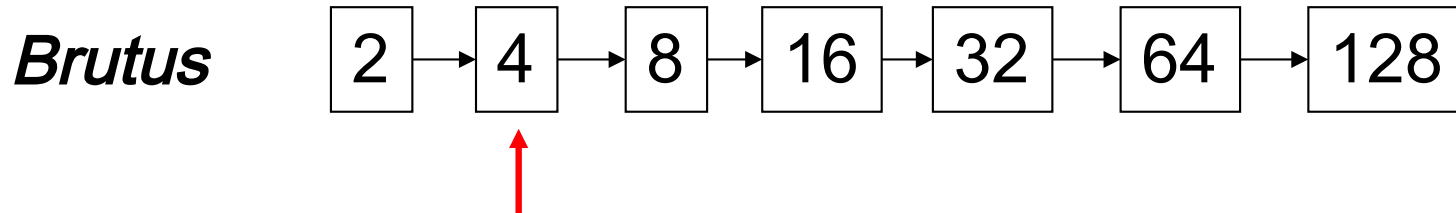


Brutus AND Caesar

2

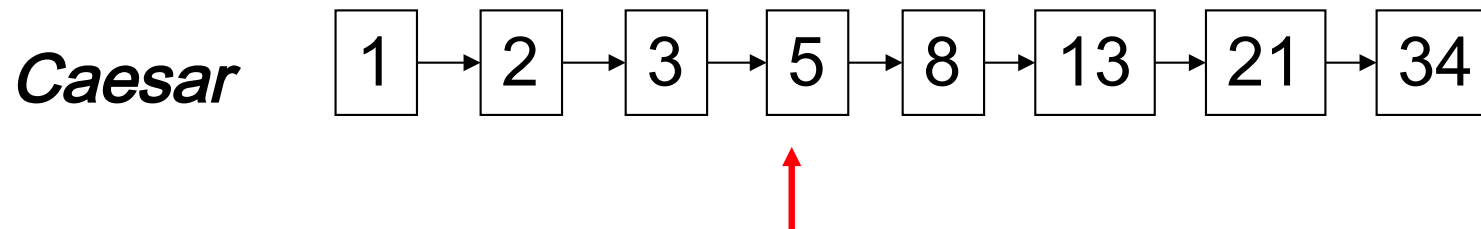
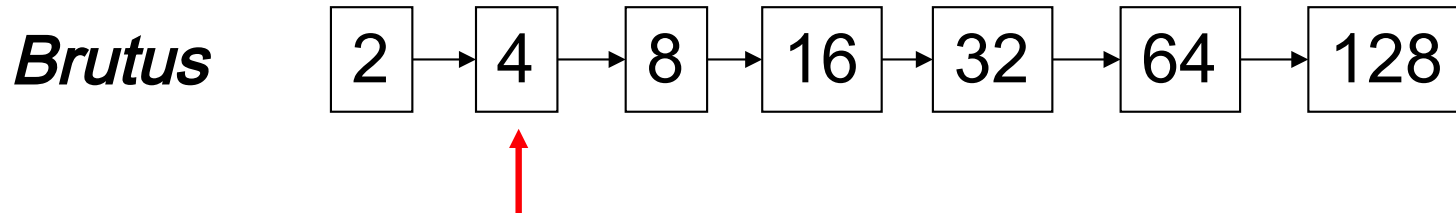
The merge

- Walk through the two postings simultaneously



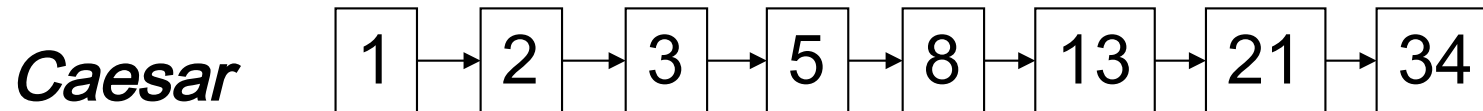
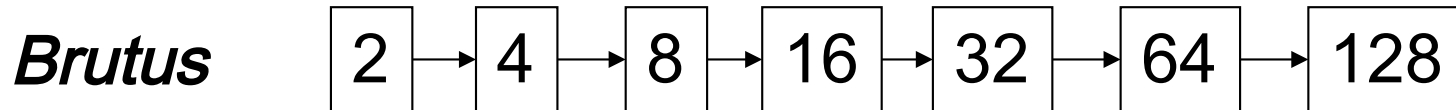
The merge

- Walk through the two postings simultaneously



The merge

- Walk through the two postings simultaneously

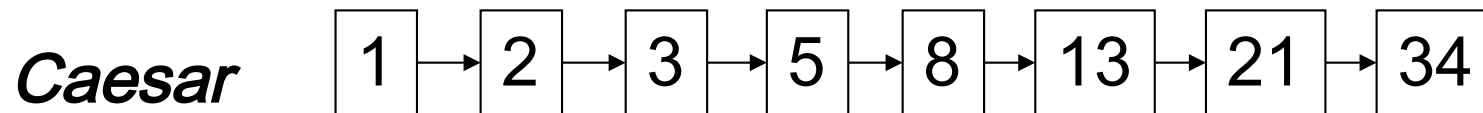
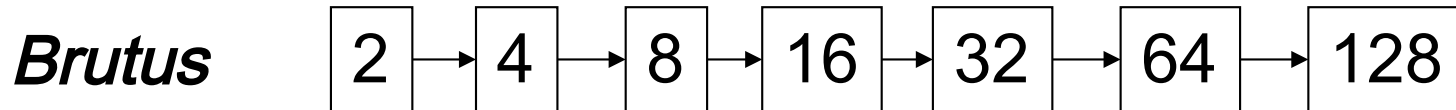


...



The merge

- Walk through the two postings simultaneously

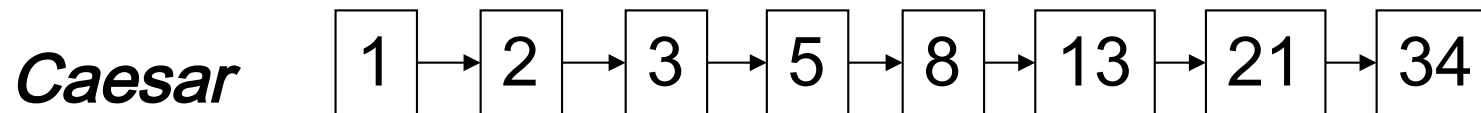
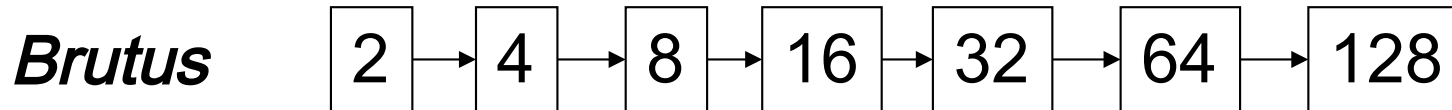


What assumption are we making about the postings lists?

For efficiency, when we construct the index, we ensure that the postings lists are sorted

The merge

- Walk through the two postings simultaneously



What is the running time?

$O(\text{length1} + \text{length2})$

Merging

What about an arbitrary Boolean formula?

(Brutus OR Caesar) AND NOT

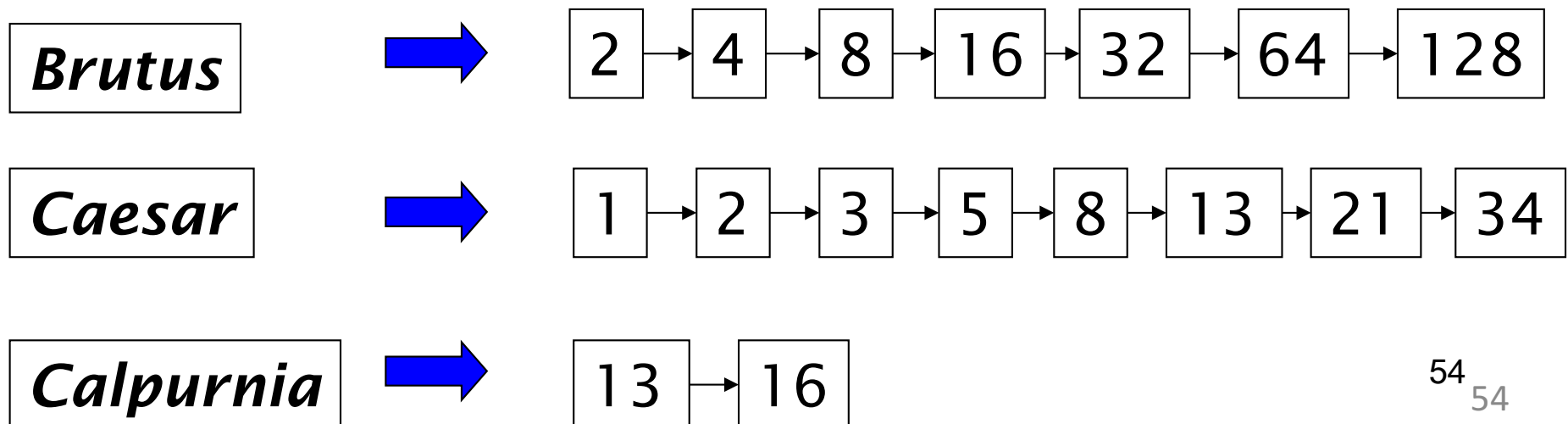
(Antony OR Cleopatra)

- $x = (\text{Brutus OR Caesar})$
- $y = (\text{Antony OR Cleopatra})$
- $x \text{ AND NOT } y$
- Is there an upper bound on the running time?
 - $O(\text{total_terms} * \text{query_terms})$
- What about *Brutus AND Calpurnia AND Caesar*?

Query optimization

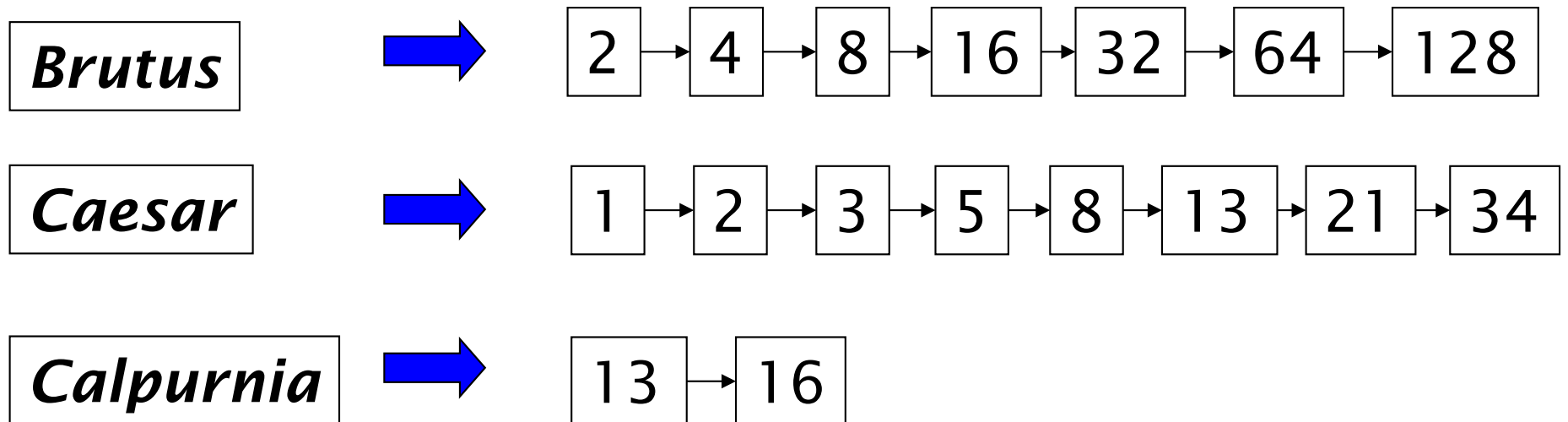
Query: *Brutus AND Calpurnia AND Caesar*

- Consider a query that is an *AND* of t terms.
- For each of the terms, get its postings, then *AND* them together
- What is the best order for query processing?



Query optimization example

- Heuristic: **Process in order of increasing freq**:
 - *merge the two terms with the shortest postings list*
 - *this creates a new AND query with one less term*
 - *repeat*

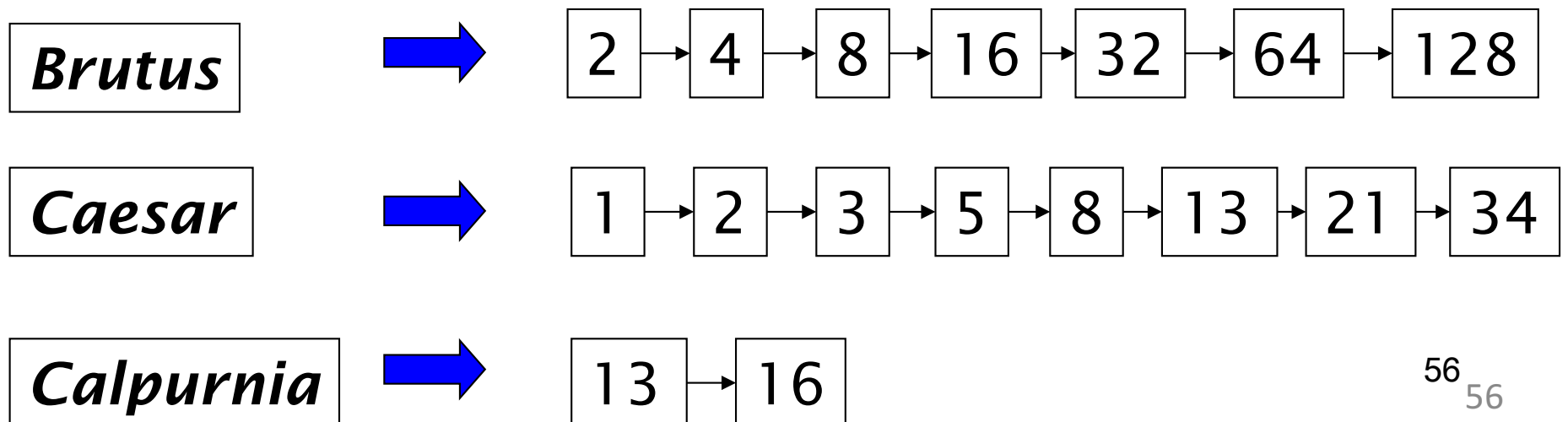


Execute the query as (*Calpurnia* **AND** *Brutus*) **AND** *Caesar*.

Query optimization

Query: *Brutus OR Calpurnia OR Caesar*

- Consider a query that is an *OR* of t terms.
- What is the best order for query processing?
- Same: still want to merge the shortest postings lists first



Query optimization in general

- (madding **OR** crowd) **AND** (ignoble **OR NOT** strife)
- Need to evaluate OR statements first
- Which OR should we do first?
 - Estimate the size of each OR by the sum of the posting list lengths
 - NOT is just the number of documents minus the length
 - Then, it looks like an AND query:
 - x **AND** y

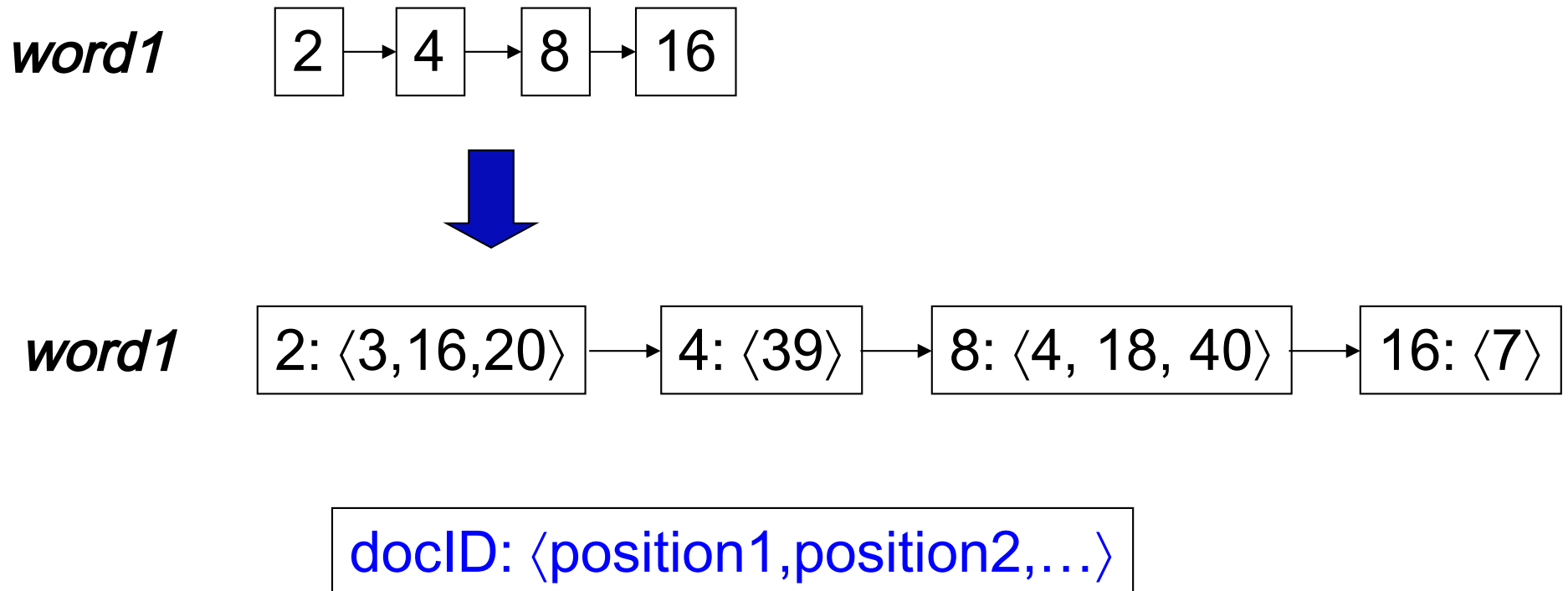
Phrase queries

- Want to be able to answer queries such as “***Michigan Tech***”
- “*I went to a college in houghton*” would not a match
 - The concept of phrase queries has proven easily understood by users
 - Many more queries are *implicit phrase queries*

How can we modify our existing postings lists to support this?

Positional indexes

- In the postings, store a list of the positions in the document where the term occurred



Positional index example

be:

1: $\langle 7, 18, 33, 72, 86, 231 \rangle$

2: $\langle 3, 149 \rangle$

4: $\langle 17, 191, 291, 430, 434 \rangle$

5: $\langle 363, 367 \rangle$

to:

1: $\langle 4, 17, 32, 90 \rangle$

2: $\langle 5, 50 \rangle$

4: $\langle 12, 13, 429, 433, 500 \rangle$

5: $\langle 4, 15, 24, 38, 366 \rangle$

1. Looking only at the “be” postings list, which document(s) could contain “***to be or not to be***”?
2. Using both postings list, which document(s) could contain “***to be or not to be***”?
3. Describe an algorithm that discovers the answer to question 2 (hint: think about our linear “merge” procedure)

Processing a phrase query: “to be”

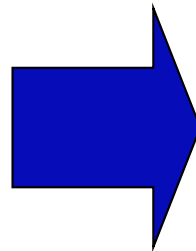
- Find all documents that have have the terms using the “merge” procedure
- For each of these documents, “merge” the position lists with the positions offset depending on where in the query the word occurs

be:

4: $\langle 17, 191, 291, 430, 434 \rangle$

to:

4: $\langle 12, 13, 429, 433, 500 \rangle$



be:

4: $\langle 17, 191, 291, 430, 434 \rangle$

to:

4: $\langle 13, 14, 430, 434, 501 \rangle$

Processing a phrase query: “to be”

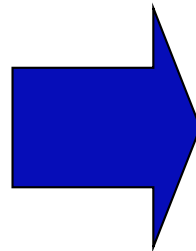
- Find all documents that have have the terms using the “merge” procedure
- For each of these documents, “merge” the position lists with the positions offset depending on where in the query the word occurs

be:

4: $\langle 17, 191, 291, 430, 434 \rangle$

to:

4: $\langle 12, 13, 429, 433, 500 \rangle$



be:

4: $\langle 17, 191, 291, 430, 434 \rangle$

to:

4: $\langle 13, 14, 430, 434, 501 \rangle$

What about proximity queries?

- Find “Michigan” within k words of “Tech”
- Similar idea, but a bit more challenging
- Naïve algorithm for merging position lists
 - Assume we have access to a merge with offset exactly i procedure (similar to phrase query matching)
 - for $i = 1$ to k
 - if merge with offset i matches, return a match
 - if merge with offset $-i$ matches, return a match
- Naïve algorithm is inefficient, but doing it efficiently is a bit tricky

Positional index size

- You can compress position values/offsets
- Nevertheless, a positional index expands postings storage *substantially*
- 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

- What does adding positional information do to the size of our index?
- Need an entry for each occurrence, not just once per document
- Posting size depends on the lengths of the documents

Positional index 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
1 000	?	
1 00,000		

Positional index 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
1 000	1	
1 00,000	?	

Positional index 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
1 000	1	?
1 00,000	1	

Positional index 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
1 000	1	1
1 00,000	1	?

Positional index 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
1 000	1	1
1 00,000	1	1 00

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

Popular phrases

- Is there a way we could speed up common/popular phrase queries?
 - ***“Barack Obama”***
 - ***“Donald Trump”***
 - ***“New York”***
- We can store the phrase as another *term* in our dictionary with it's own postings list
- This avoids having to do the “merge” operation for these frequent phrases

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.

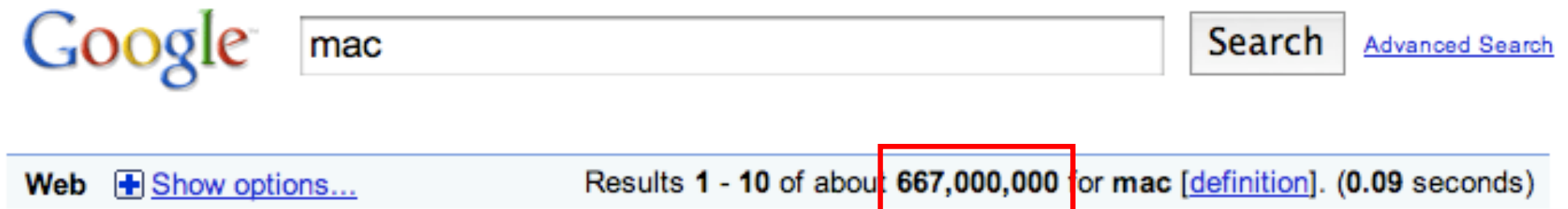
Problem with Boolean search: feast or famine

- 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

Scoring as the basis of ranked retrieval

We want to return in order the documents most likely to be useful to the searcher

Assign a score that 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
- Besides whether or not a query occurs in a document, what other indicators may be useful?
 - **How many times the word appears**
 - **Where the word appears**
 - **How “important” the word is**
- 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
- $\text{jaccard}(A, B) = |A \cap B| / |A \cup B|$
- $\text{jaccard}(A, A) = 1$
- $\text{jaccard}(A, B) = 0$ 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 matrix

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

What information is lost with this representation?

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 in this course.
- 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.

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 high 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
- 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.

idf example, suppose $N = 1$ million

term	df_t	idf_t
calpurnia	1	6
animal	100	4
sunday	1,000	3
fly	10,000	2
under	100,000	1
the	1,000,000	0

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

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

Effect of idf on ranking

- 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**.

tf-idf weighting

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

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

- **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**

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 $\in \mathbb{R}^{|V|}$

Documents as vectors

- So 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

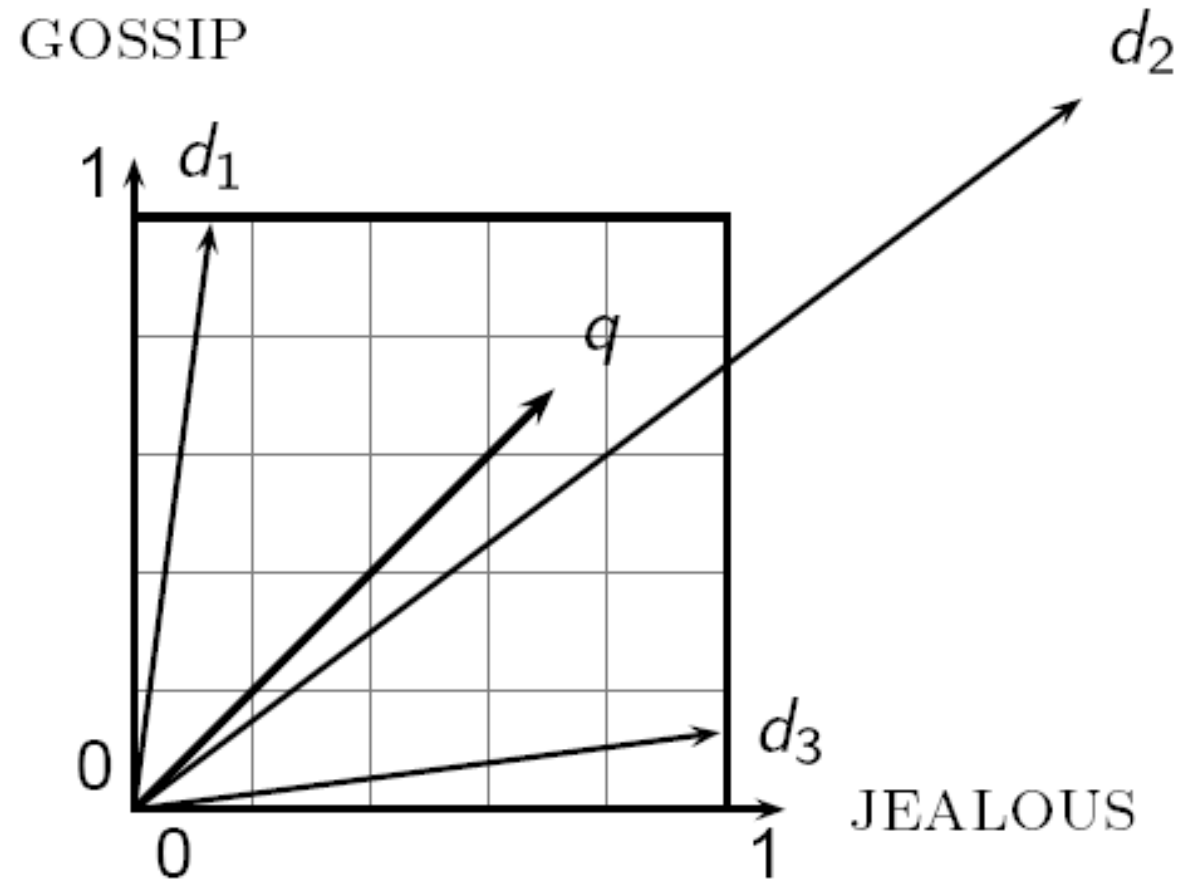
- Key idea 1: Do the same for queries: represent them as vectors in the space
- Key idea 2: Rank documents according to their proximity to the query in this space
- proximity = similarity of vectors
- proximity \approx inverse of distance
- Recall: We do this because we want to get away from the you're-either-in-or-out 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 \vec{q} and \vec{d}_2 is large even though the distribution of terms in the query \vec{q} and the distribution of terms in the document \vec{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 $\cos(\text{angle}(\text{query}, \text{document}))$
- Cosine is a monotonically decreasing function for the interval $[0^\circ, 180^\circ]$

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)

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

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):

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

for q, d length-normalized.

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

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

$$0.789 \times 0.832 + 0.515 \times 0.555 + 0.335 \times 0.0 + 0.0 \times 0.0$$

$$\approx 0.94$$

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

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

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

Example 2

- Documents: 8 Wikipedia articles, 4 about TMNT Leonardo, Raphael, Michelangelo, and Donatello, and 4 about the painters with same



Query: “Raphael is cool but rude, Michelangelo is a party dude”

##		but	cool	dude	party	michelangelo	raphael	rude	dist
##	1tmnt.txt	5	0	0	1	3	7	0	231.3374
##	2tmnt.txt	20	0	0	0	4	24	0	320.2842
##	3tmnt.txt	8	0	3	2	66	20	0	323.7993
##	4tmnt.txt	17	1	0	0	9	67	1	272.6610
##	5real.txt	1	0	0	0	0	0	0	178.2835
##	6real.txt	16	0	0	0	8	6	0	694.5725
##	7real.txt	12	0	0	0	113	4	0	679.4814
##	8real.txt	33	0	0	0	14	84	0	508.7386
##	query.txt	1	1	1	1	1	1	1	0.0000

Document Lengths and Normalization

Different documents have different lengths. Total word counts:

##	1tmnt	2tmnt	3tmnt	4tmnt	5real	6real	7real	8real	query
##	2078	2943	3133	2625	1420	6250	5046	4842	7

- Wikipedia entry on Leonardo the painter is > 2x as long as that on Leonardo the TMNT (6250 vs 2943 words). The query is only 7 distinct words long.
- The count vectors X_d and Y should be normalized
 - document length normalization: divide X by its sum

$$X \leftarrow X / \sum_{w=1}^W X_w$$

- L2 length normalization: divide X by its L2 length

$$X \leftarrow X / \|X\|_2$$

Back to Example

##		dist/doclen	dist/l2len
##	1tmnt.txt (tmnt don)	0.3912246	1.395611
##	2tmnt.txt (tmnt leo)	0.3873645	1.373594
##	3tmnt.txt (tmnt mic)	0.3801707	1.330071
##	4tmnt.txt (tmnt rap)	0.3785931	1.317932
##	5real.txt (real don)	0.3980160	1.412714
##	6real.txt (real leo)	0.3922210	1.402623
##	7real.txt (real mic)	0.3920400	1.362544
##	8real.txt (real rap)	0.3823314	1.343668
##	query.txt (tmnt don)	0.0000000	0.0000000

- Normalized form
- What about some words being more helpful than others? **Common words**, especially, are not going to help find relevant documents

All together

- For a document-term matrix

	word 1	word 2	...	word W
doc 1				
doc 2				
\vdots				
doc D				

- Normalization – scale each row by something (sum or L2 norm)
- IDF weighting – scale each column by something (multiply by $\log(D / n_w)$)

Back to Example

##			dist/dl/idf	dist/l2/idf
##	1tmnt.txt	(tmnt don)	0.8462084	3.030274
##	2tmnt.txt	(tmnt leo)	0.8376958	2.981983
##	3tmnt.txt	(tmnt mic)	0.8218870	2.886687
##	4tmnt.txt	(tmnt rap)	0.8183885	2.859938
##	5real.txt	(real don)	0.8611110	3.067073
##	6real.txt	(real leo)	0.8484139	3.046022
##	7real.txt	(real mic)	0.8480160	2.958339
##	8real.txt	(real rap)	0.8266119	2.916465
##	query.txt	(tmnt don)	0.0000000	0.000000

- Didn't work as well as hoped. Why?

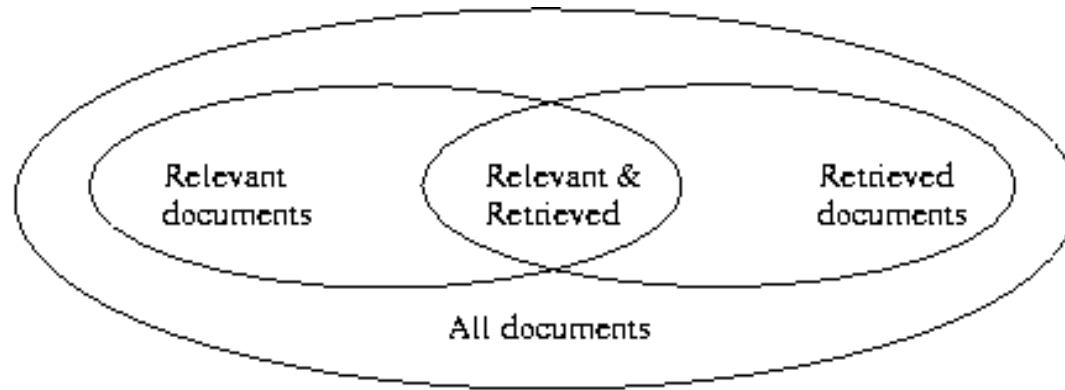
Feedback

- Use people to confirm the relevance of what has been found
- Rocchio's algorithm takes feedback from the user about relevance, and then refines the query and repeats the search
 - User gives an initial query Y
 - Computer returns documents it believes to be relevant, and user divides these into sets: relevant R and not relevant NR
 - Computer updates the query string as

$$Y \leftarrow \alpha Y + \frac{\beta}{|R|} \sum_{X_d \in R} X_d - \frac{\gamma}{|NR|} \sum_{X_d \in NR} X_d$$

- Repeat steps 2 and 3

Evaluating Text Retrieval



- **Precision:** the percentage of retrieved documents that are in fact relevant to the query (i.e., “correct” responses)

$$precision = \frac{|\{Relevant\} \cap \{Retrieved\}|}{|\{Retrieved\}|}$$

- **Recall:** the percentage of documents that are relevant to the query and were, in fact, retrieved

$$recall = \frac{|\{Relevant\} \cap \{Retrieved\}|}{|\{Relevant\}|}$$

Precision vs. Recall

	Truth:Relvant	Truth:Not Relevant
Algorithm:Relevant	TP	FP
Algorithm: Not Relevant	FN	TN

- We've been here before!

- $\text{Precision} = \text{TP} / (\text{TP} + \text{FP})$
- $\text{Recall} = \text{TP} / (\text{TP} + \text{FN})$
- Trade off:
 - If algorithm is 'picky': precision high, recall low
 - If algorithm is 'relaxed': precision low, recall high

		actual outcome	
		1	0
predicted outcome	1	<i>a</i>	<i>b</i>
	0	<i>c</i>	<i>d</i>

- BUT: recall often hard if not impossible to calculate

Precision Recall Curves

- If we have a labelled training set, we can calculate recall.
- For any given number of returned documents, we can plot a point for precision vs. recall. (similar to thresholds in ROC curves)
- Different retrieval algorithms might have very different curves - hard to tell which is “best”

