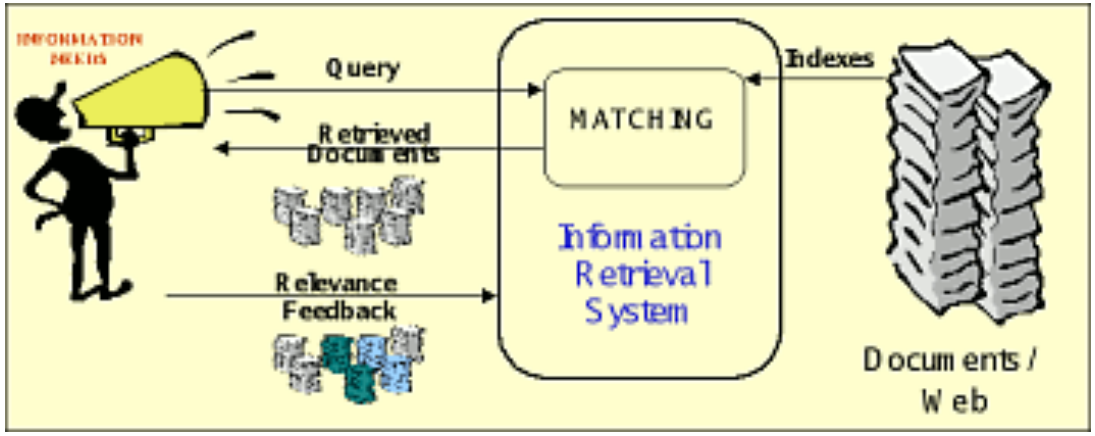
Information Retrieval: An Intro

Ishwar K Sethi

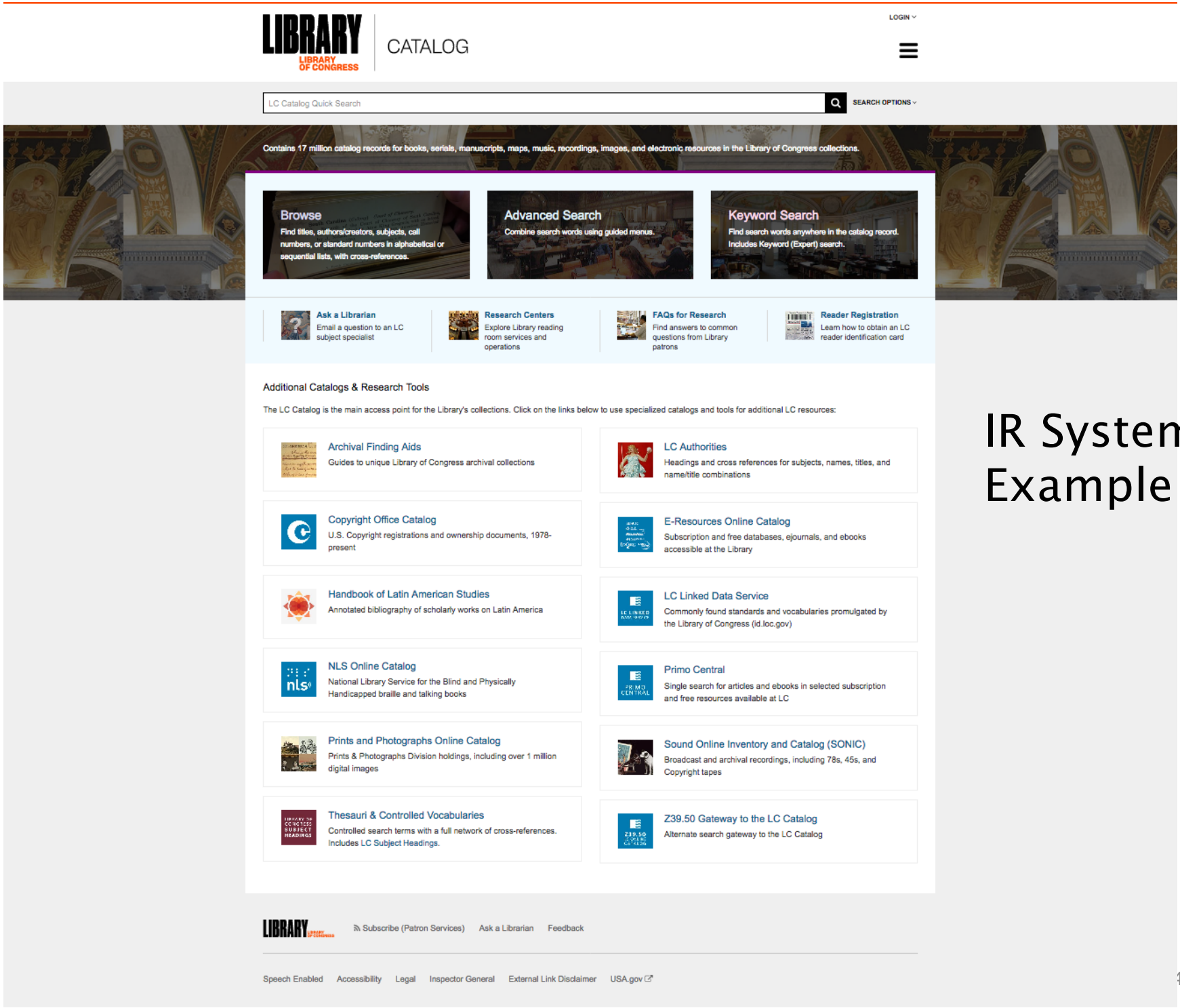
What is Information Retrieval?

- Information retrieval (IR) is a field of study dealing with representation, storage, organization of, and access to documents. The documents may be books, reports, pictures, videos, web pages or multimedia files. The whole point of an IR system is to provide a user easy access to documents containing the desired information.

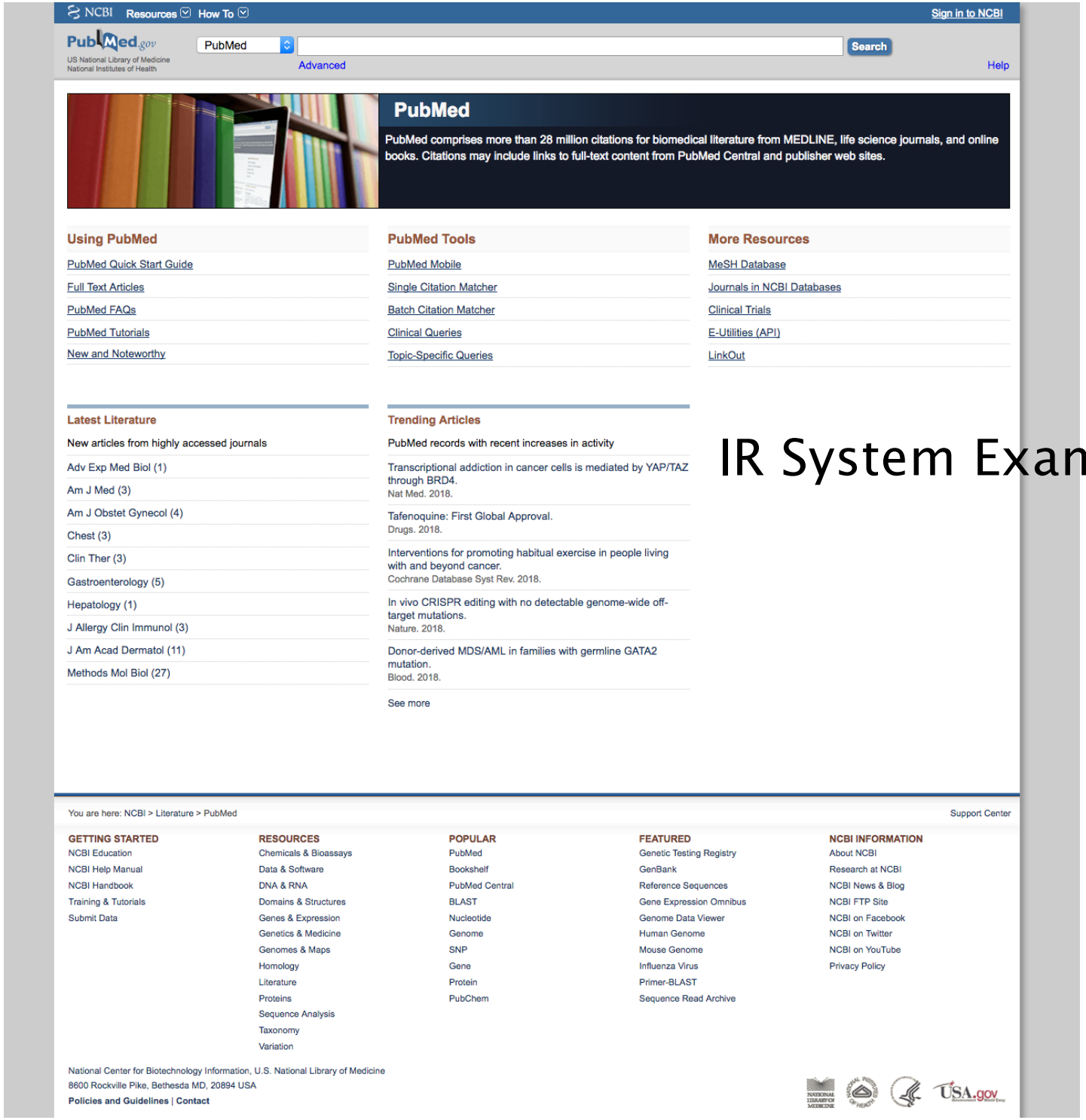


Information vs Data Retrieval

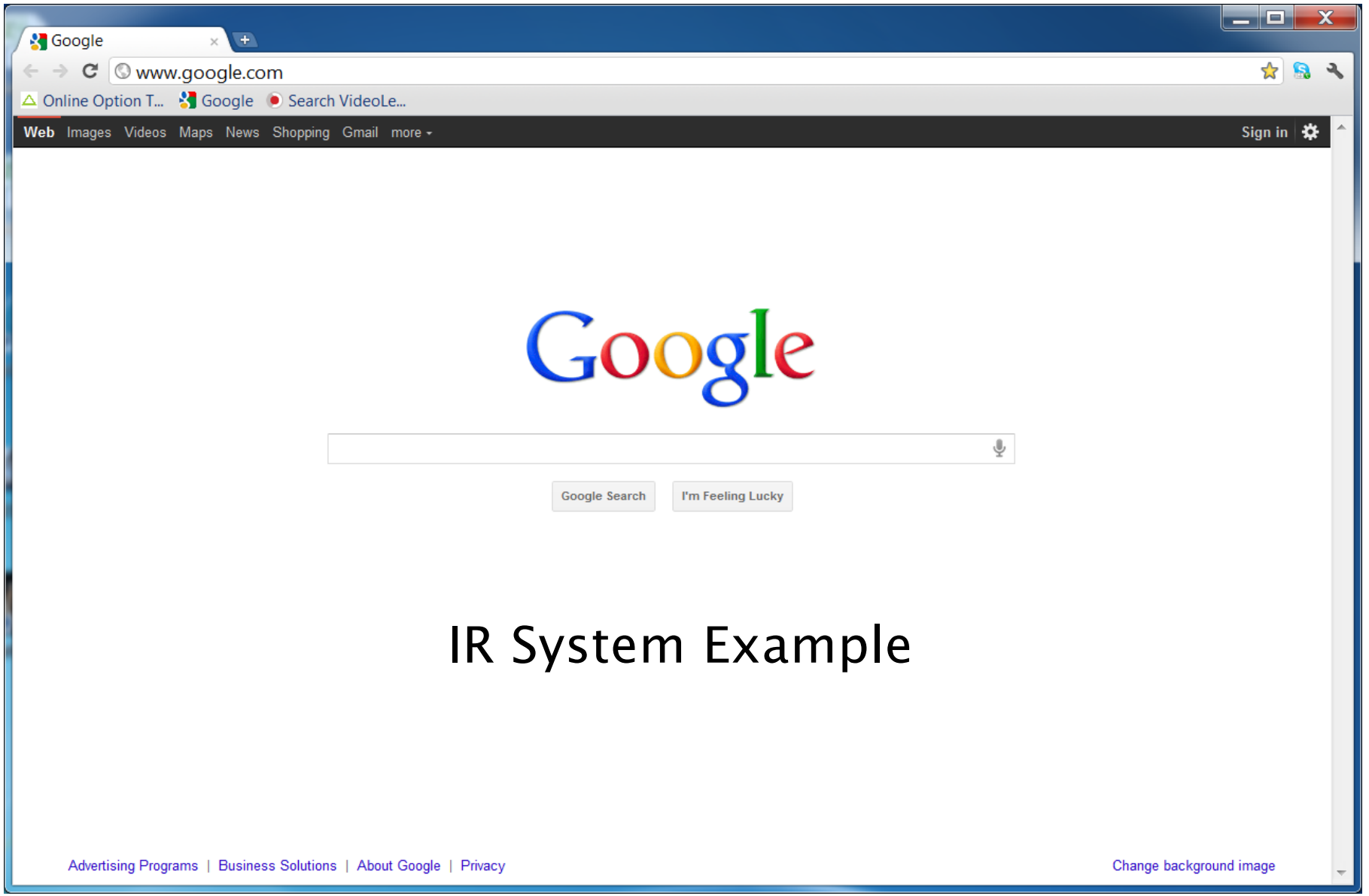
- IR deals with unstructured/semi-structured data while DBMS deals with structured data with well-defined semantics
- Querying an IR system produces multiple results with ranking. Partial match is allowed
- Querying a DBMS system produces exact/precise results or no results if no exact match is found



IR System Example



IR System Example



IR System Example

Documents in IR

- Text documents (For example newspaper reports, scientific articles, email messages)

- Graphical and Multimedia documents (For example images, line drawings, videos, PowerPoint presentations, and web pages)

- Spoken documents (Voice messages, radio news and programs, podcasts, telephone conversations)

Document Characterization

- *Metadata* characterization
  - This kind of characterization refers to ownership, authorship and other items of information about a document. The Library of Congress subject coding is also an example of metadata. Another example of metadata is the category headings at Internet search engine *Yahoo*. To standardize category headings, many areas use specific *ontologies*, which are hierarchical taxonomies of terms describing certain knowledge topics.

Document Characterization

- *Presentation* characterization.
  - This refers to attributes that control the formatting or presentation of a document.

Document Characterization

- *Content* characterization.
  - This refers to attributes that denote the semantic content of a document. **Content characterization is of primary interest in IR**.

Document Representation in IR

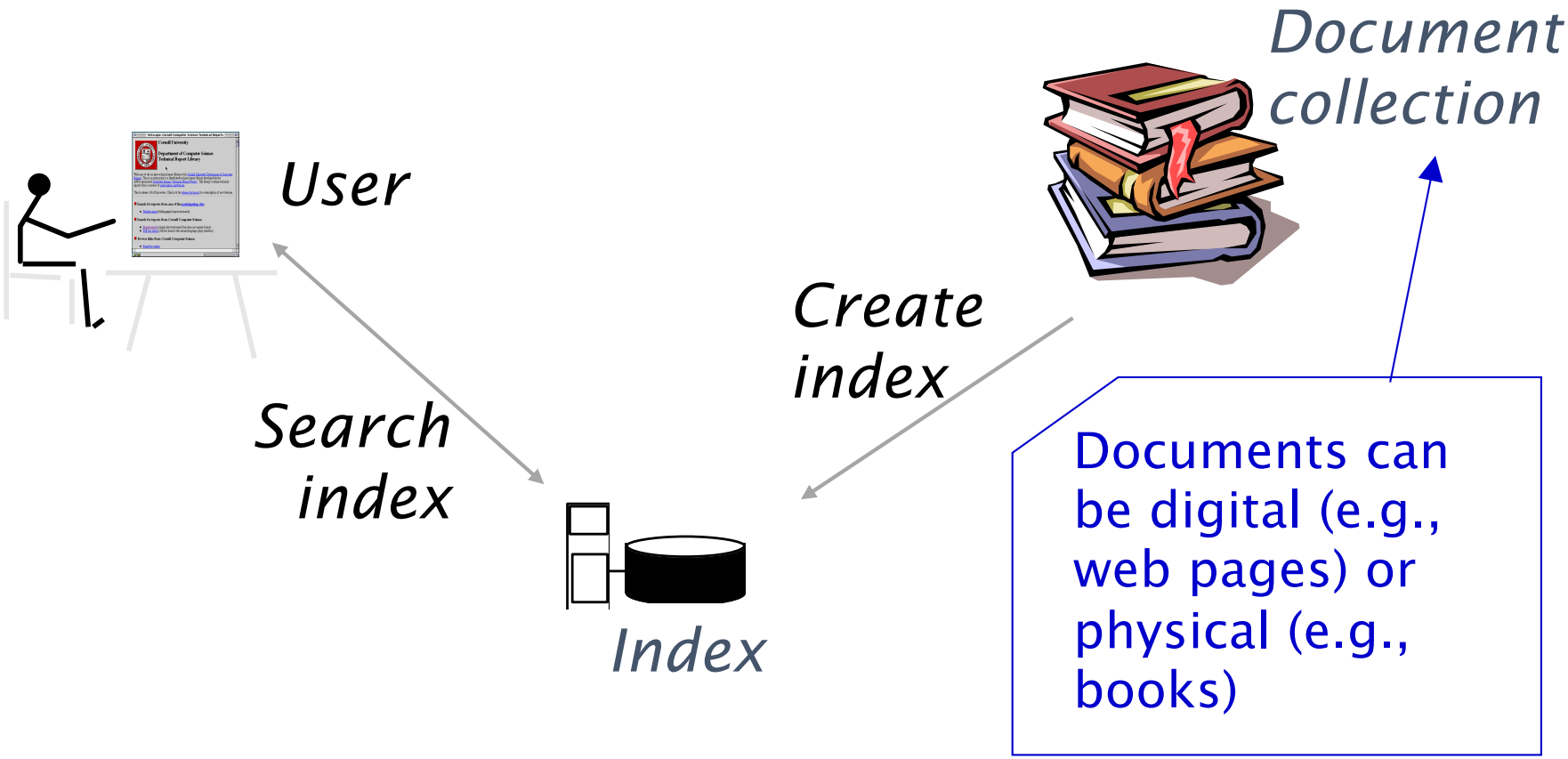
- The common practice in IR is to represent a textual document by a bag of words/set of keywords called index terms or simply terms.
- An *index term* is a word or a phrase in a document whose semantics give an indication of the document's theme. The index terms, in general, are mainly nouns because nouns have meaning by themselves.

Document Representation in IR

- It is not uncommon to look at images/multimedia documents also in terms of words using the Bag of Word (BoW) representation.

Indexes

IR systems rely on indexes while searching for documents to satisfy a user's query.



Boolean Model of IR

- The *Boolean* model of a document uses a set of index terms with binary weights. If the weight of a term is one (zero), the term is present (absent) in the document.
- To retrieve documents from an IR system using Boolean modeling, the query is constructed of index terms linked by three connectives: *not, and, or*. Only those documents are retrieved that are ‘true’ for the query.
- Primary commercial retrieval tool for 3 decades.
- Many search systems still use Boolean model:
  - Email, library catalog, Mac OS X Spotlight

Term-document Matrix in BM

	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

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. (incidence vectors)
- To answer query: take the vectors for ***Brutus, Caesar*** and ***Calpurnia*** (complemented) ➔ bitwise *AND*.
- 110100 *AND* 110111 *AND* 101111 = 100100.

The term-document matrix representation is useful for understanding the document model but not good enough for as a vehicle for implementation. Think about a million documents and 1k terms.

Ranking in Boolean Model

- In the Boolean model, the documents are either *relevant (true)* or *non-relevant (false)*; there is no notion of a partial match or ranking.
- A slight modification of the full Boolean search is sometimes preferred. The modification allows only AND logic but takes account of the actual *number* of terms the query has in common with a document. This number is known as the *co-ordination level*. The search strategy is often called *simple matching*. Because at any level we can have more than one document, the documents are said to be *partially* ranked by the co-ordination levels.

Ranking in Boolean Model

	D1	D2	D3	D4
K1	1	1	1	1
K2	1	1	0	0
K3	1	1	0	1
K4	1	0	0	0

$Q = K1$ AND $K2$ AND $K3$

Result using the coordination level

3 $D1$, $D2$
2 $D4$
1 $D3$

$$| D \cap Q |$$

The size of the overlap between D and Q, each represented as a set of keywords

Matching Functions for Ranking

Dice's coefficient: $\frac{2|D \cap Q|}{|D|+|Q|}$

Jaccard's coefficient: $\frac{|D \cap Q|}{|D \cup Q|}$

Cosine coefficient: $\frac{|D \cap Q|}{|D|^{1/2} \times |Q|^{1/2}}$

Overlap coefficient: $\frac{|D \cap Q|}{\min(|D|,|Q|)}$

Proximity Operators in Boolean Matching

- Instead of simply searching for the presence/absence of the query keywords, proximity operators allow adding additional constraints. For example, the two query keywords should occur within a space of few words.

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

Westlaw: Largest commercial (paying subscribers) legal search service. Tens of terabytes of data; 700,000 users

Vector Space Model (VSM)

- In this model documents and queries are represented in a high-dimensional vector space. The vector space is defined by indexing terms and the location of a document in the vector space is determined by the weights assigned to indexing terms for the document.
- The VSM can be considered a general version of the BM with non-binary weights. The main advantage of non-binary weights is that partial matching is possible which allows retrieved documents to be ranked in terms of the match score.

Cosine Similarity Measure

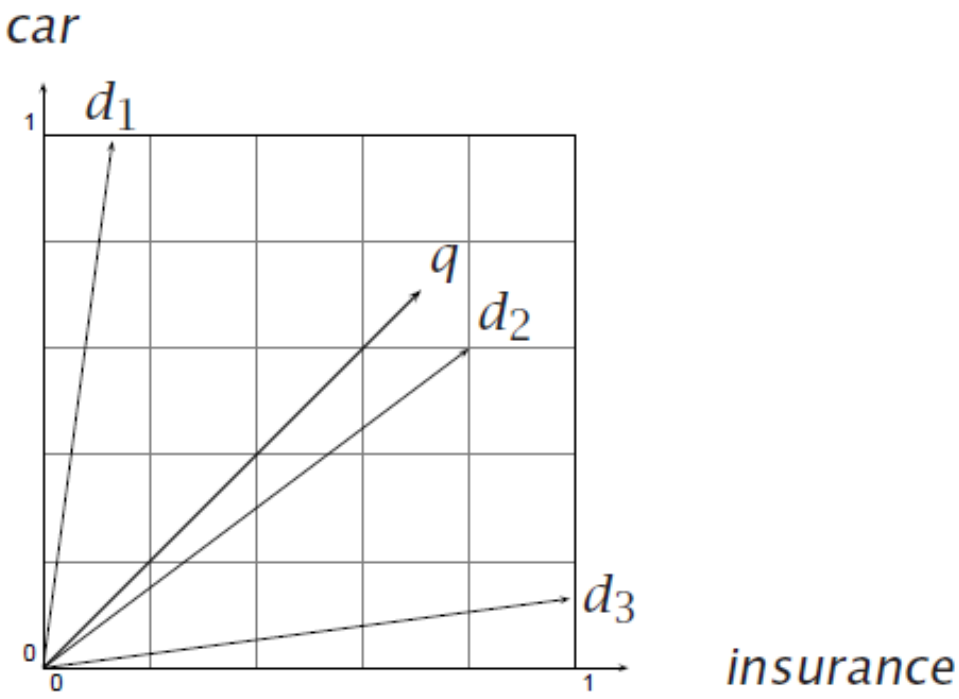
$\vec{d}_j$ vector representation of document D_j

$\vec{q}$ Query vector

$$\cos(\vec{d}_j, \vec{q}) = \frac{\sum_{i=1}^{n} d_{ij} q_i}{\sqrt{\sum_{i=1}^{n} d_{ij}^2} \sqrt{\sum_{i=1}^{n} q_i^2}}$$

The cosine similarity is also known as *normalized correlation coefficient*. For the same query, different documents give rise to different similarity values to yield a ranking of documents for the query.

The Vector Space Model (normalized vectors)



How Do We Weigh Terms?

- Simplest term (vector component) weightings are:
  - count of number of times word occurs in document
  - binary: word does or doesn't occur in document
- However, general experience is that a document is a better match if a word occurs three times than once, but not a three times better match.
- This leads to a series of weighting functions that damp the term weighting, e.g., $1 + \log(x)$, $x > 0$, or $\sqrt{x}$.
- This is a good thing to do, but still imperfect: it doesn't capture that the occurrence of a term in a document is more important if that term does not occur in many other documents.

Term Weights

tf_{ij} *Term frequency*

Number of occurrences of term or word W_i in document D_j.

df_i *Document frequency*

Number of documents in the collection containing word w_i.

cf_i *Collection frequency*

Total number of occurrences of word W_i in the collection of N documents.

$$\text{weight}(i,j) = \begin{cases} (1 + \log(\text{tf}_{i,j})) \log \frac{N}{\text{df}_i} & \text{if } \text{tf}_{i,j} \geq 1 \\ 0 & \text{if } \text{tf}_{i,j} = 0 \end{cases}$$

Inverse Document Frequency

tf-idf weighting

- The weighting scheme shown earlier is known as the tf-idf term weighting as it is the product of term's tf weight and its idf weight.
- 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
- The term weight increases with the number of occurrences within a document
- The term weight increases with the rarity of the term in the collection

TF-IDF Example:

Number of Documents: 806,791

Term	Document Frequency	Calculated IDF Value
Car	18,165	1.65
Auto	6,723	2.08
Insurance	19,241	1.62
Best	25,235	1.5

=LOG10(806791/18165)

Term frequencies for three documents

	Doc1	Doc2	Doc3
Car	27	4	24
Auto	3	33	0
Insurance	0	33	29
Best	14	0	17

TF-IDF Example:

Calculated term weights for three documents

	Doc1	Doc2	Doc3
Car	4.06	2.64	3.92
Auto	3.07	5.24	0
Insurance	0	4.08	3.99
Best	3.22	0	3.35

=(1+LOG10(27))*LOG10(806791/18165)

Tf*Idf

Cosine (D1,D2) ?

=(4.06*2.64+3.07*5.24)/
(SQRT(4.06^2+3.07^2+0+3.22^2))*SQRT(2.64^2+5.24^2+4.08^2+0)
=0.623

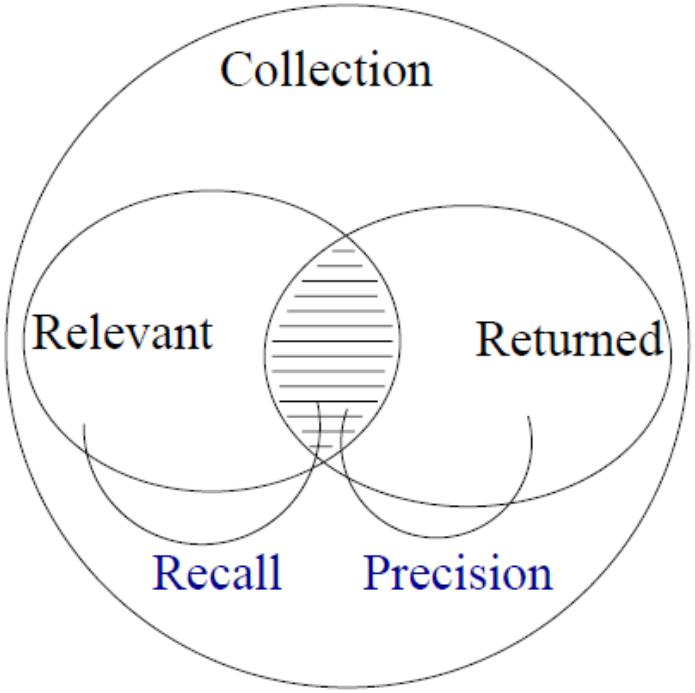
Summary – Vector Space Ranking

- Represent the query as a weighted tf-idf vector
- Represent each document as a weighted tf-idf vector
- Compute the cosine similarity score for the query vector and each document vector
- Rank documents with respect to the query by score
- Return the top *K* (e.g., *K* = 10) to the user

How good are the retrieved docs?

- *Precision* : Fraction of retrieved docs that are relevant to user's information need
- *Recall* : Fraction of relevant docs in collection that are retrieved

Accuracy is not a useful measure when the target set is a tiny fraction of the total set.



Evaluation of *ranked* results	Ranking 1	Ranking 2	Ranking 3
	d1: ✓	d10: ×	d6: ×
	d2: ✓	d9: ×	d1: ✓
	d3: ✓	d8: ×	d2: ✓
	d4: ✓	d7: ×	d10: ×
	d5: ✓	d6: ×	d9: ×
	d6: ×	d1: ✓	d3: ✓
	d7: ×	d2: ✓	d5: ✓
	d8: ×	d3: ✓	d4: ✓
	d9: ×	d4: ✓	d7: ×
	d10: ×	d5: ✓	d8: ×
precision at 5	1.0	0.0	0.4
precision at 10	0.5	0.5	0.5

We generally consider top-k documents to measure precision and recall

The *F* Measure

Combines precision and recall into a single measure

$$F = \frac{1}{\alpha\frac{1}{P} + (1-\alpha)\frac{1}{R}}$$

where P is precision, R is recall and α weights precision and recall. (Or in terms of β, where $\alpha = 1/(\beta^2 + 1)$.)

A value of $\alpha = 0.5$ is often chosen.

$$F = \frac{2PR}{R + P}$$

F measure is also known as *harmonic mean*

Index Terms

Table 1 Common Words in *Tom Sawyer*

Word	Freq.	Use
the	3332	determiner (article)
and	2972	conjunction
a	1775	determiner
to	1725	preposition, verbal infinitive marker
of	1440	preposition
was	1161	auxiliary verb
it	1027	(personal/expletive) pronoun
in	906	preposition
that	877	complementizer
he	877	(personal) pronoun
I	783	(personal) pronoun
his	772	(possessive) pronoun
you	686	(personal) pronoun
Tom	679	proper noun
with	642	preposition

Total word count: 71,370

Number of different words: 8,018

The above two numbers suggest that words in the corpus occur 'on an average' about 9 times. Of course some words occur lot more than 9 times and some occur fewer than 9 times. In fact Table 2 provides a frequency of frequencies information.

Table 2 on next slide

Index Terms

Frequencies of frequencies in Tom Sawyer

Word Frequency	Frequency of Frequency
1	3993
2	1292
3	664
4	410
5	243
6	199
7	172
8	131
9	82
10	91
11–50	540
51–100	99
> 100	102

Is this group of words important?

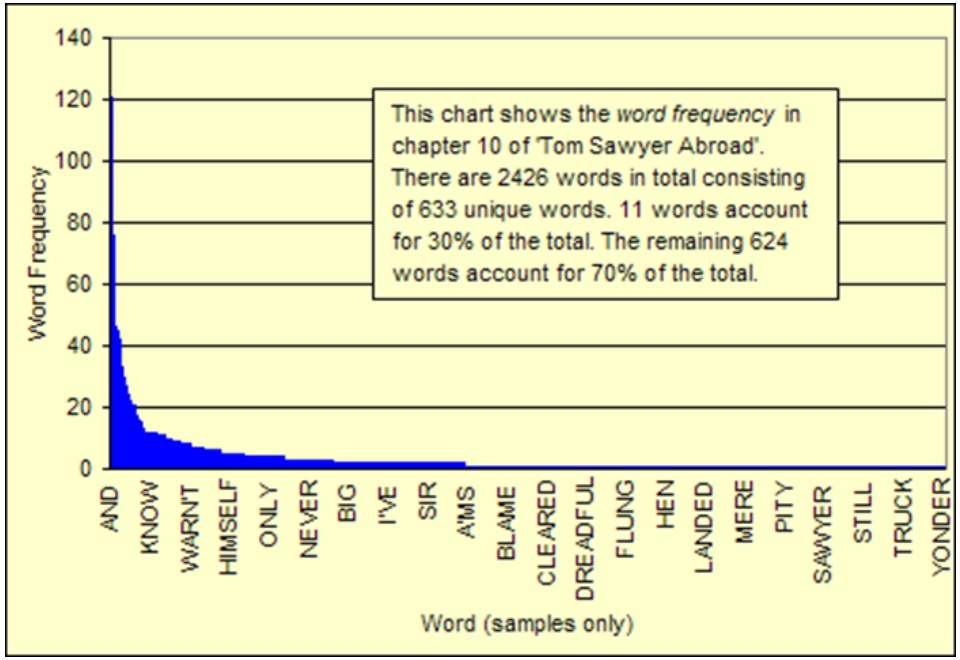
Is this group of words important?

-The most common 100 words account for slightly over 50% of the word tokens in the text.

-Almost half of the words occur only once in the text.

Zipf's Law

The law states that there is an inverse relation between the frequency of a word and its rank. This means that the product of frequency and rank is a constant.

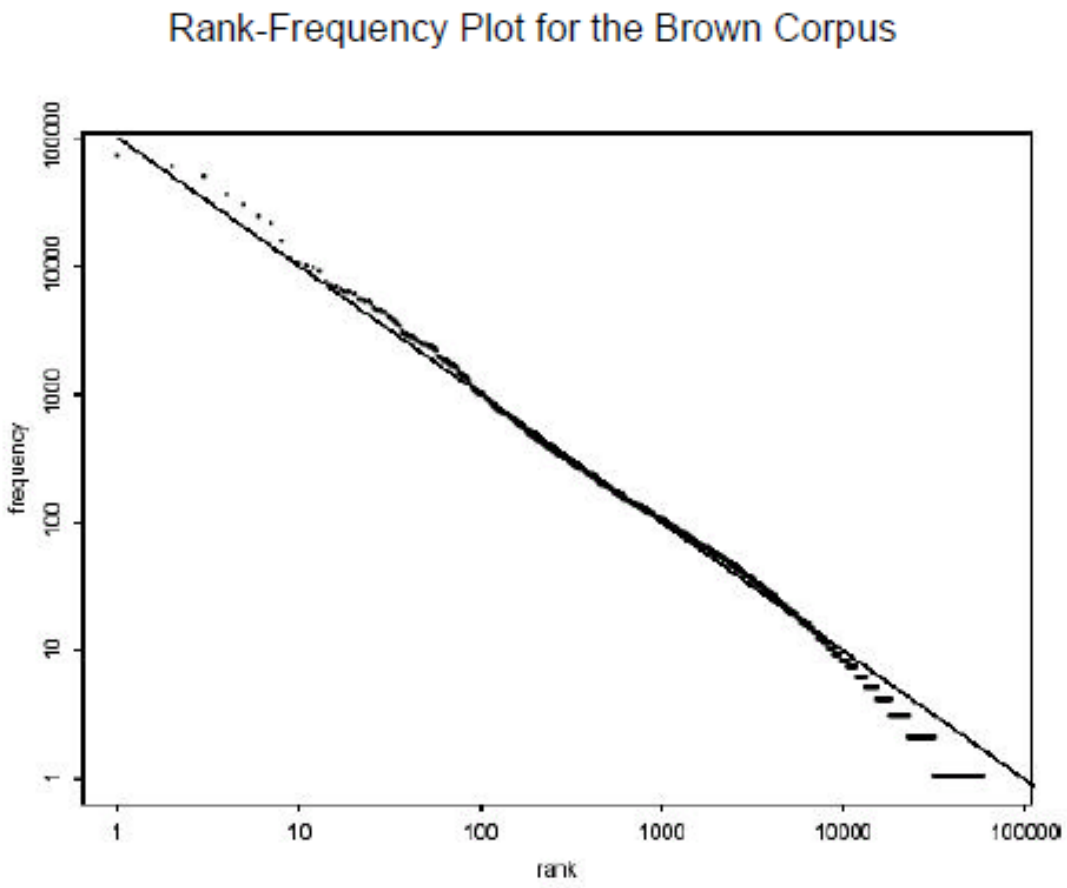


Zipf's law in Tom Sawyer

Word	Freq. (*f*)	Rank (*r*)	*fxr*
the	3332	1	3332
and	2972	2	5944
a	1775	3	5235
he	877	10	8770
but	410	20	8400
be	294	30	8820
there	222	40	8880
one	172	50	8600
about	158	60	9480
more	138	70	9660
never	124	80	9920
Oh	116	90	10440
two	104	100	10400
turned	51	200	10200
you'll	30	300	9000
name	21	400	8400
comes	16	500	8000
group	13	600	7800
lead	11	700	7700
friends	10	800	8000
begin	9	900	8100
family	8	1000	8000
brushed	4	2000	8000
sins	2	3000	6000
Could	2	4000	8000
Applausive	1	8000	8000

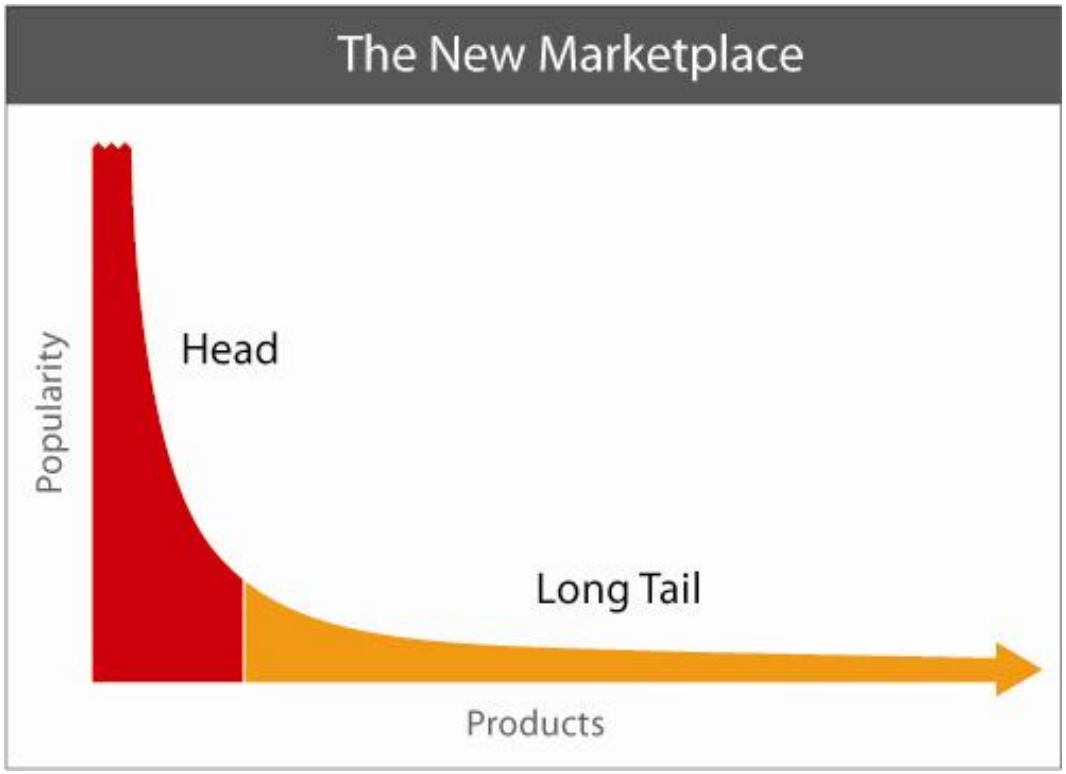
Rank-Frequency Plot

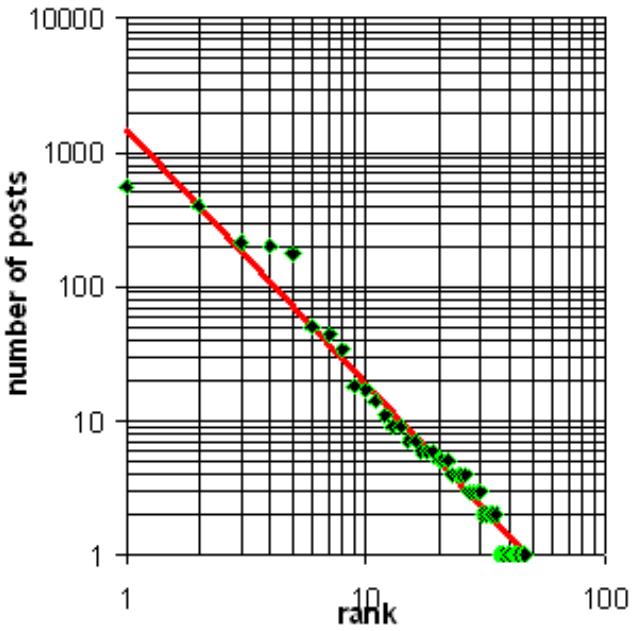
Log-log plot is linear



Zipf's Law Examples

The presence of Zipf's law is found in many areas.
Zipf's law is an example of power law.





Term Distribution Model

Poisson Distribution

$$p(k;\alpha) = \frac{\alpha^k}{k!}e^{-\alpha}$$

where α is the average number of occurrences of the event being

model and $p(k; \alpha)$ is the probability that k occurrences of the event

will occur. Both the mean and the variance of the Poisson distribution

are equal to α.

Poisson distribution is a standard probabilistic model that is used for modeling events such as the number of defective items returned in a given period of time, the number of typing mistakes on a page, and the number of cars pulling in at a gas station in a given time.

Example: A switchboard receives on the average 16 calls per minute. If the switchboard can handle at most 24 calls per minute, what is the probability that in any one minute the switchboard will saturate?

Solution: Saturation will occur if more than 24 calls are received in any minute. Thus the probability of saturation is:

$$P[sat] = \sum_{k=25}^{\infty} \alpha^k \frac{e^{-\alpha}}{k!}$$

$$= \sum_{k=25}^{\infty} 16^k \frac{e^{-16}}{k!} \cong 0.017 \cong 1/60$$

Thus only once in every 60 minutes will a caller experience saturation.

To see how well the Poisson distribution captures terms distribution, consider the following results for four terms in the *New York* corpus having $N = 79291$ documents.

=23533/79291

Term	Doc. Freq	Coll. Freq	α	$N(1 - p(0; \alpha)$
follows	21744	23533	0.2968	20363
transformed	807	840	0.0106	835
soviet	8204	35337	0.4457	28515
students	4953	15925	0.2008	14425

The Poisson distribution fits well for *non-content terms but not so well for* content terms.

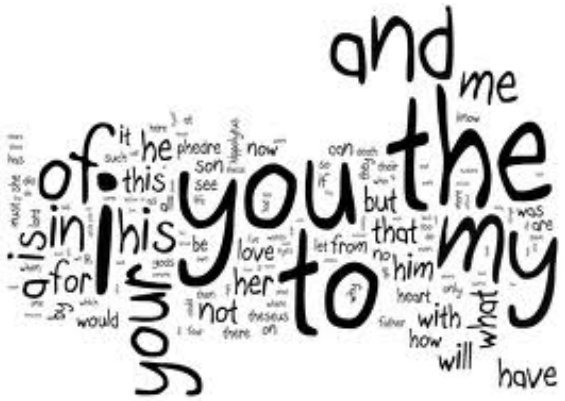
The observation that the Poisson distribution does not fit well for content words can be used to advantage. We can compare a term's frequency with its Poisson estimate and use the difference between the two to arrive at the term weight. This approach is known as *residual inverse document frequency (RIDF)*. The residual is defined as follows:

$$RIDF = IDF - \log_2 \frac{1}{1 - p(0;\alpha)} = IDF + \log_2 (1 - p(0;\alpha))$$

where IDF = $\log_2$(N/df).

Stop Words

Not all words are content words. The most frequent words are *function words* that are unlikely to aid searching. It is thus common to process the input text for stop word removal by using a stop list similar to the one shown below.



Stemming

Stemming is another common operation that is performed prior to index weight computation. *Stemming* refers to removing one or more suffixes off a word to reduce it to its root form. For example, *compression*, *compressed*, and *compressing* are all reduced to *compress*. *Porter's* stemming algorithm is the most widely algorithms. It should be noted that stemming might create occasional problems by truncating two semantically different words.

http://tartarus.org/~martin/PorterStemmer/

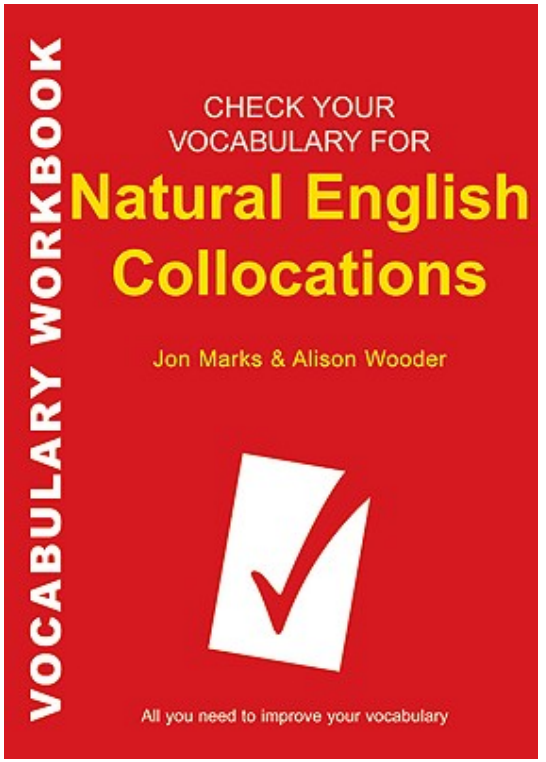
Not much used now a days

Case folding

Case folding means replacing all uppercase letters with lowercase letters. Thus, the strings *The*, *the*, and *THE* would all be treated as *the*.

Phrases

- Often two or more words are used as a phrase to convey special meaning. Some examples are *disk drive*, *below the belt, weapons of mass destruction* etc.
- Such phrases are good candidates for index terms
- These phrases are also called *collocations*



Finding Phrases

- One simple approach is to look for frequent word pairs by computing bigram frequencies. Such word pairs are then filtered using the parts of speech combinations such as “noun noun” or “adjective noun”

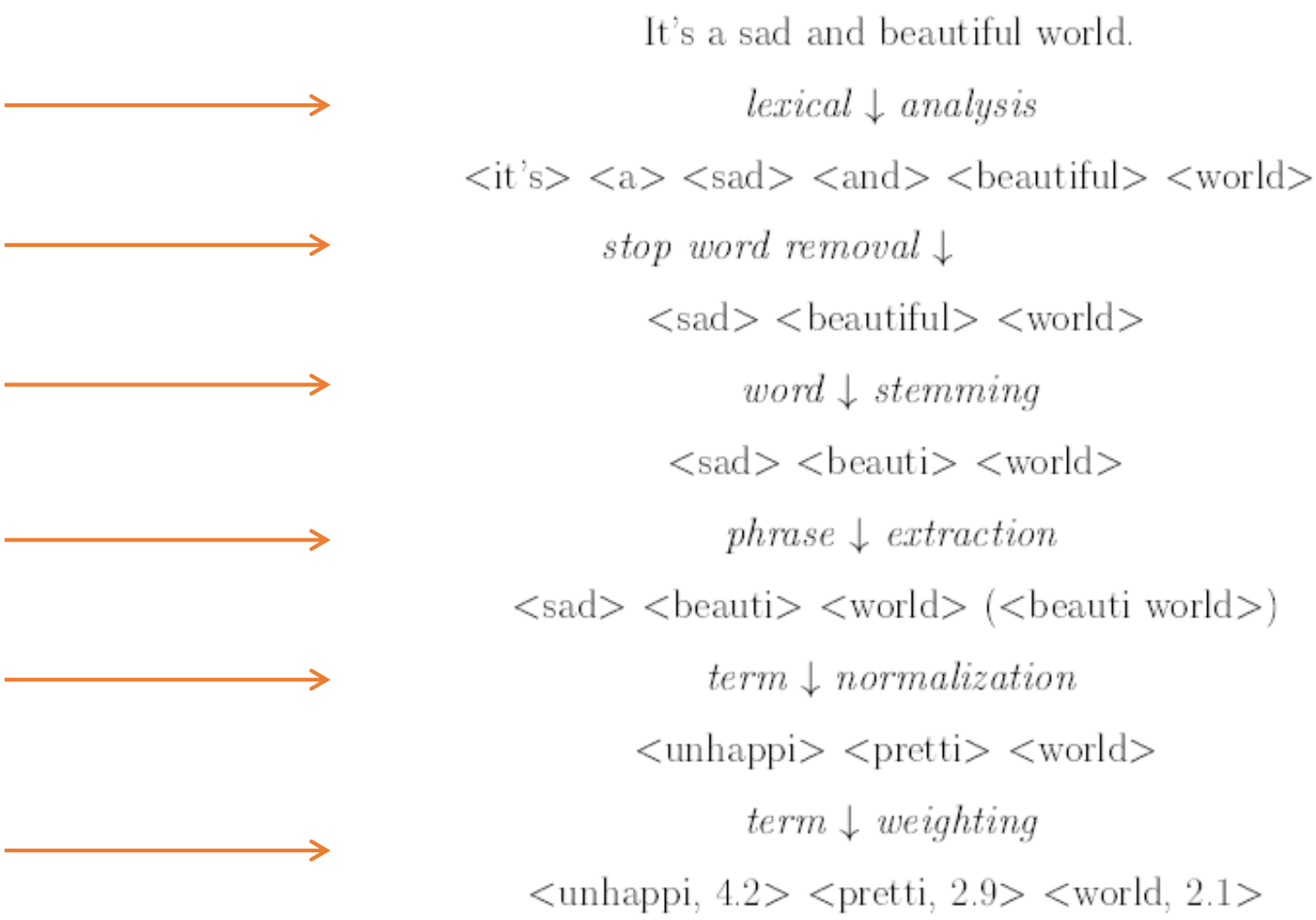
Filtered common bigrams in the NYT

Frequency	Word 1	Word 2	POS pattern
11487	New	York	A N
7261	United	States	A N
5412	Los	Angeles	N N
3301	last	year	A N
3191	Saudi	Arabia	N N
2699	last	week	A N
2514	vice	president	A N
2378	Persian	Gulf	A N
2161	San	Francisco	N N
2106	President	Bush	N N
2001	Middle	East	A N
1942	Saddam	Hussein	N N
1867	Soviet	Union	A N
1850	White	House	A N
1633	United	Nations	A N
1337	York	City	N N
1328	oil	prices	N N
1210	next	year	A N
1074	chief	executive	A N
1073	real	estate	A N

Term Normalization

- Needed to deal with
  - Synonymy (Different words implying the same concept, for example *car*, *auto*, *automobile*)
  - Done by using a controlled vocabulary

Steps Towards Index Generation



Bigger collections

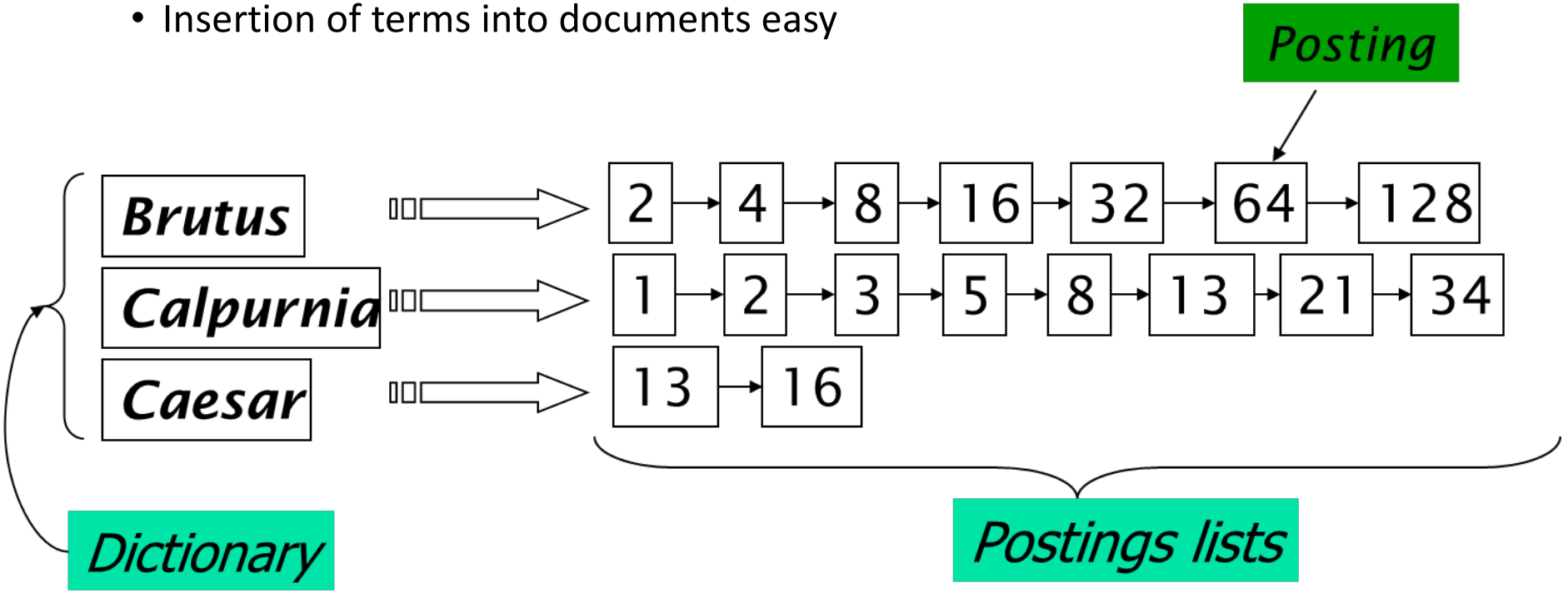
- Consider *N* = 1 million documents, each with about 1000 words.
- Avg. 6 bytes/word including spaces/punctuation
  - 6GB of data in the documents.
- Say there are *M* = 500K *distinct* terms among these.

Document Matrix Representation Not Suitable

- 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 store a list of all documents that contain *T*.
  - Identify each by a **docID**, a document serial number
- Linked list structure is generally used for this purpose.
  - Dynamic space allocation
  - Insertion of terms into documents easy



Term frequency & location information is also stored in many systems

Indexer Steps

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

Doc 1

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

Doc 2

Term	Doc #
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

Sort by terms

Merge multiple term entries in a single document and add frequency information

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

Split the result into a *Dictionary* file and a *Postings* file

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

Term	N docs	Coll freq
ambitious	1	1
be	1	1
brutus	2	2
capitol	1	1
caesar	2	3
did	1	1
enact	1	1
hath	1	1
I	1	2
i'	1	1
it	1	1
julius	1	1
killed	1	2
let	1	1
me	1	1
noble	1	1
so	1	1
the	2	2
told	1	1
you	1	1
was	2	2
with	1	1

Doc #	Freq
2	1
2	1
1	1
2	1
1	1
1	1
2	2
1	1
1	1
2	1
1	2
1	1
2	1
1	1
1	2
2	1
1	1
2	1
2	1
1	1
2	1
2	1
2	1
1	1
2	1
2	1

Ndocs is the number of documents with the term

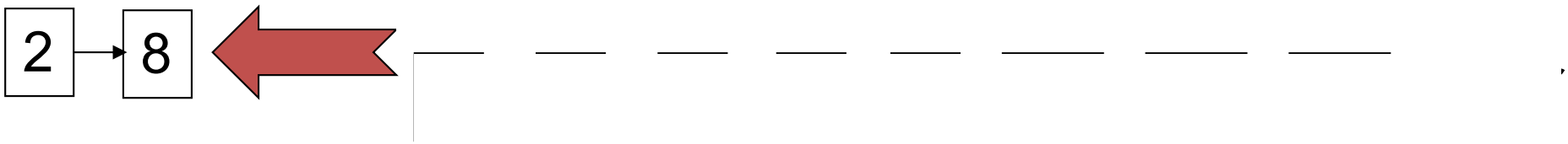
Retrieval with Inverted Lists

- Consider processing the query:

  Brutus *AND* ***Caesar***

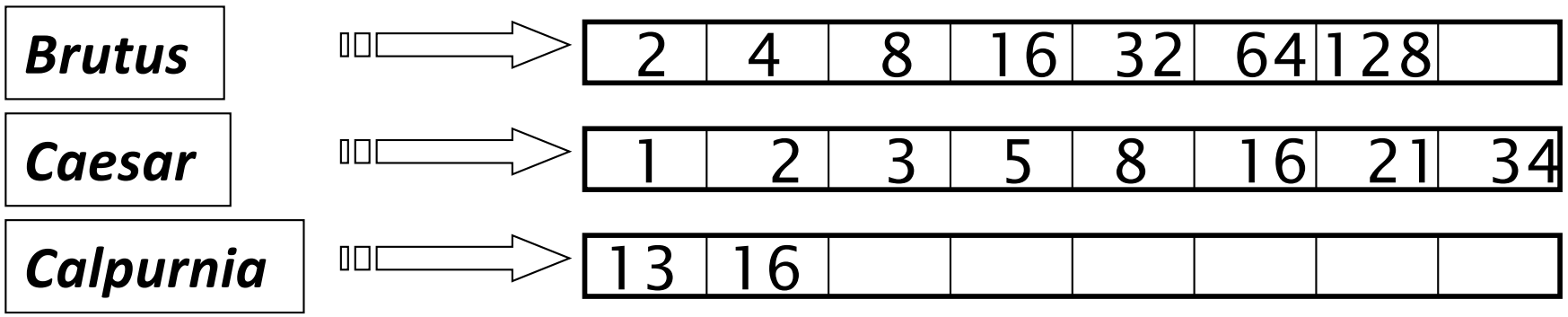
  - Locate ***Brutus*** in the Dictionary;
    - Retrieve its postings.
  - Locate *Caesar* in the Dictionary;
    - Retrieve its postings.
  - "Merge" the two postings:

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



Query optimization

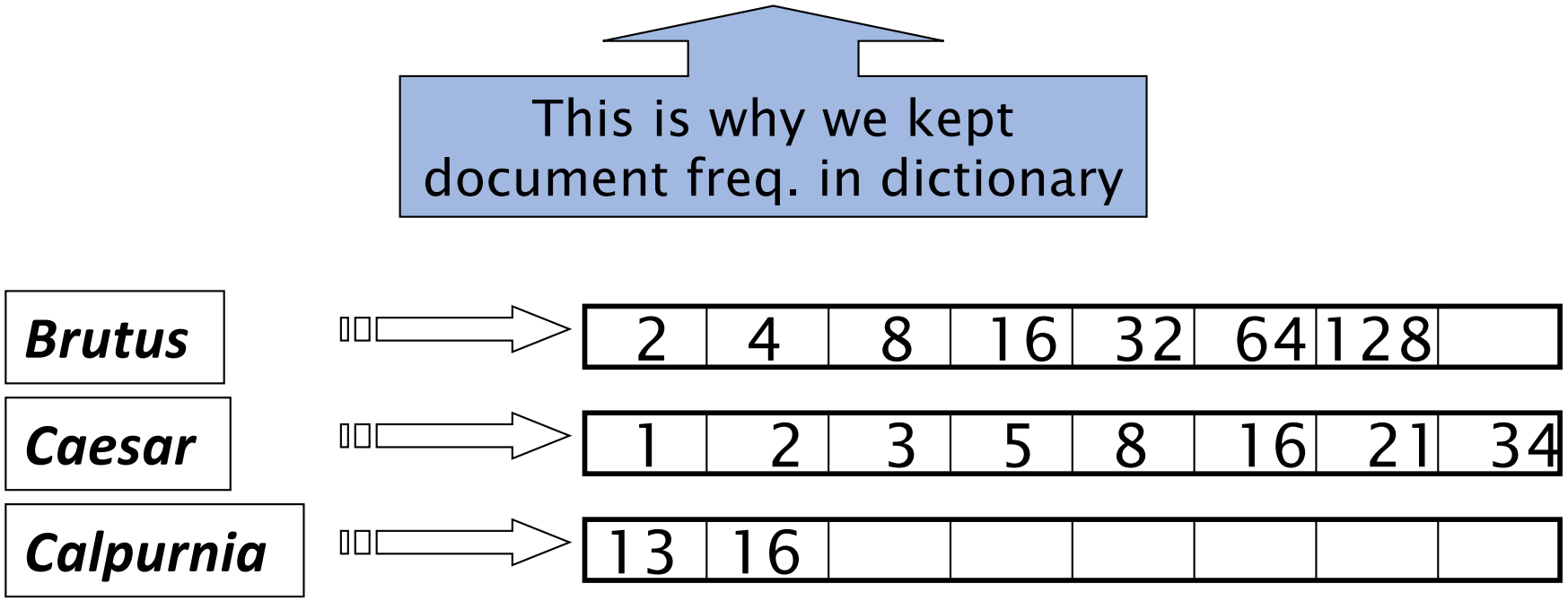
- Consider a query that is an *AND* of *n* terms.
- For each of the *n* terms, get its postings, then *AND* them together.
- Is their any advantage in certain term order for query processing?



Query: ***Brutus*** *AND* ***Calpurnia*** *AND* ***Caesar***

Query optimization example

- Process in order of increasing freq:
  - *start with smallest set, then keep cutting further*.



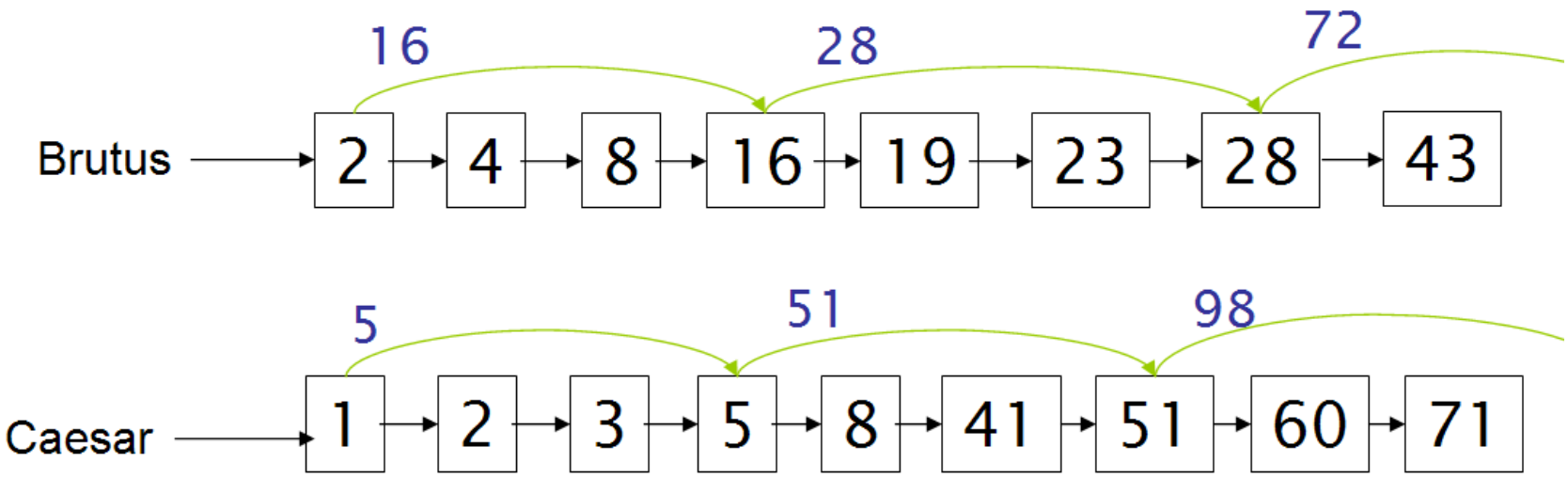
Execute the query as (***Calpurnia*** *AND* ***Brutus)*** *AND* ***Caesar***.

More general optimization

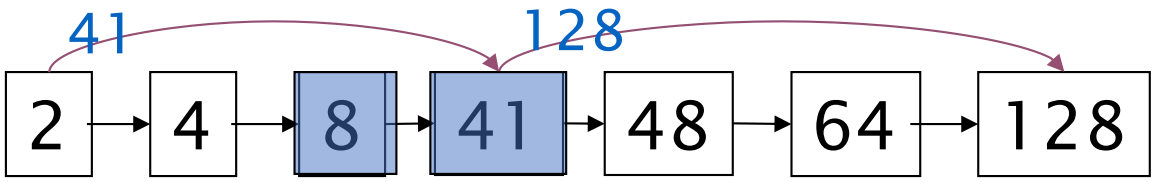
- e.g., *(**madding** OR **crowd**) AND (**ignoble** OR **strife**)*
- Get doc. freq.'s for all terms.
- Estimate the size of each *OR* by the sum of its doc. freq.'s (conservative).
- Process in increasing order of *OR* sizes.

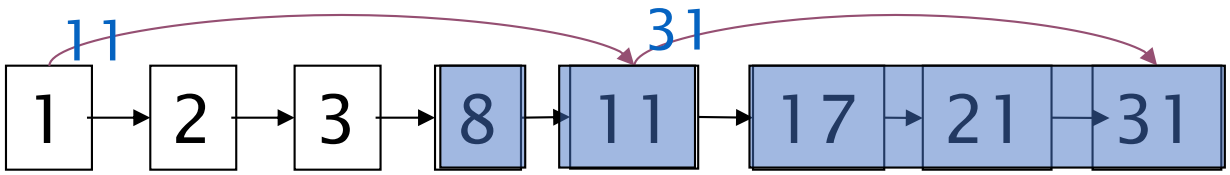
Other Improvements in Inverted Lists

- Use of *skip pointers*
- Dictionary compression
- Posting file compression



Query processing with skip pointers





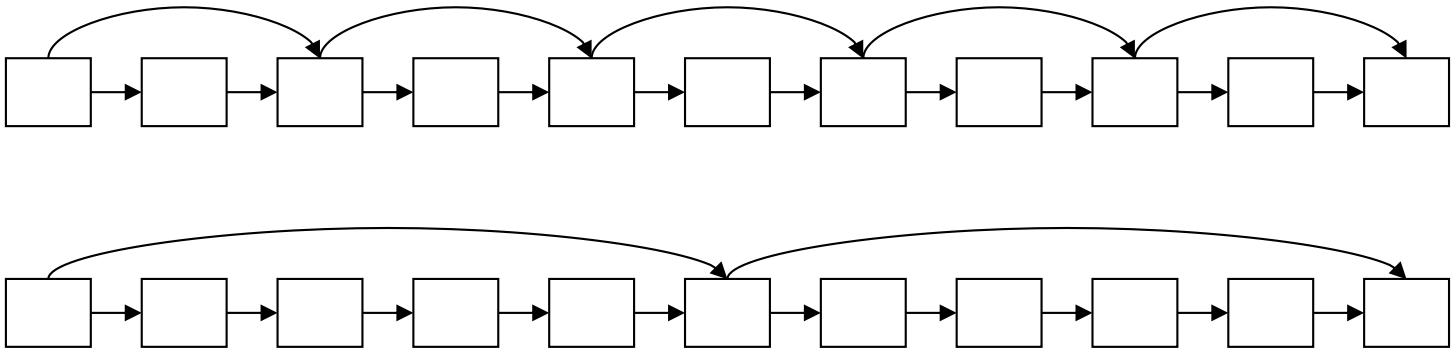
Suppose we've stepped through the lists until we process **8** on each list. We match it and advance.

We have **41** and **11** on skip pointers. **11** is smaller.

But the skip successor of **11** on the lower list is **31**, so we can skip ahead past the intervening postings.

Where do we place skips?

- Tradeoff:
  - More skips $\rightarrow$ shorter skip spans $\Rightarrow$ more likely to skip. But lots of comparisons to skip pointers.
  - Fewer skips $\rightarrow$ few pointer comparison, but then long skip spans $\Rightarrow$ few successful skips.



Looking ahead

- Document Clustering
  - Given a collection of documents, group them into coherent groups
- Document Classification
  - Given a set of topics, plus a new doc *D*, decide the topic(s) *D* represents.
- Similarity Search
- Search Engines and Link Analysis
- Latent Semantic Indexing
- Recommendation Systems

A Small Example of Text Processing

Text Processing Illustration

This example uses PlaintextCorpusReader to read text files from a folder and then performs various steps to obtain the tf-idf representation for each read file

```
import numpy as np
import nltk
import pandas as pd
```

```
from nltk.corpus.reader import PlaintextCorpusReader
mycorpus = PlaintextCorpusReader(r"/Users/isethi/Desktop/shorttexts", r".*\.txt")
print(mycorpus.fileids())
```

```
['first.txt', 'second.txt', 'third.txt']
```

```
print(mycorpus.raw())
```

```
The sky is blue and beautiful.
Love this blue and beautiful sky!The quick brown fox jumps over the lazy dog.
The brown fox is quick and the blue dog is lazy!The brown fox is quick and the blue dog is lazy!
The dog is lazy but the brown fox is quick!
```

```
type(mycorpus.raw())
```

```
str
```

```
Some operations examples
```

```
print (" ".join( mycorpus.sents('first.txt')[0]))# Joins the words of the first sentence [0] with spacing
print ("*".join( mycorpus.sents('first.txt')[1]))# Joins the words of the first sentence [0] with * as specified
```

```
The sky is blue and beautiful .
Love*this*blue*and*beautiful*sky*!
```

```
for fileid in mycorpus.fileids():
    mywords=mycorpus.words((fileid))
print(mywords)
```

```
['The', 'brown', 'fox', 'is', 'quick', 'and', 'the', ...]
```

```
print(mycorpus.sents()[0])
print(mycorpus.sents()[1])
print(mycorpus.sents()[2])
```

```
['The', 'sky', 'is', 'blue', 'and', 'beautiful', '.']
['Love', 'this', 'blue', 'and', 'beautiful', 'sky', '!']
['The', 'quick', 'brown', 'fox', 'jumps', 'over', 'the', 'lazy', 'dog', '.']
```

```
print(mycorpus.sents('first.txt'))
print(mycorpus.sents(mycorpus.fileids()[2])[1])# Prints the second sentence of the third txt file
```

```
[['The', 'sky', 'is', 'blue', 'and', 'beautiful', '.'], ['Love', 'this', 'blue', 'and', 'beautiful', 'sky', '!']]
['The', 'dog', 'is', 'lazy', 'but', 'the', 'brown', 'fox', 'is', 'quick', '!']
```

```
indx = 0
list1 = []
for fileid in mycorpus.fileids():
    mylist = mycorpus.raw(fileid)
    indx=indx+1
    list1.append(mylist)
```

```
corpus = np.array(list1)
print(corpus.shape)
print(corpus.size)
```

```
(3,)
3
```

```
# Lets convert to lower case and tokenize.
from sklearn.feature_extraction.text import CountVectorizer
vect = CountVectorizer()
#vect = CountVectorizer(stop_words='english')
vect.fit(corpus)
print(vect.get_feature_names())
print("Vocabulary size: {}".format(len(vect.vocabulary_)))
print("Vocabulary content:\n {}".format(vect.vocabulary_))
```

```
['and', 'beautiful', 'blue', 'brown', 'but', 'dog', 'fox', 'is', 'jumps', 'lazy', 'love', 'over', 'quick', 'sky', 'the', 'this']
Vocabulary size: 16
Vocabulary content:
 {'the': 14, 'sky': 13, 'is': 7, 'blue': 2, 'and': 0, 'beautiful': 1, 'love': 10, 'this': 15, 'quick': 12, 'brown': 3, 'fox': 6, 'jump
s': 8, 'over': 11, 'lazy': 9, 'dog': 5, 'but': 4}
```

```
X = vect.transform(corpus)
print(X.shape)
print(X.toarray())
```

```
(3, 16)
[[2 2 2 0 0 0 0 1 0 0 1 0 0 2 1 1]
 [1 0 1 2 0 2 2 2 1 2 0 1 2 0 4 0]
 [1 0 1 2 1 2 2 4 0 2 0 0 2 0 4 0]]
```

```
# Lets look at the BoW representation for our first document
print(vect.inverse_transform(X)[0])
```

```
['and' 'beautiful' 'blue' 'is' 'love' 'sky' 'the' 'this']
```

```
# Perform Stop Word Filtering
vect = CountVectorizer(stop_words='english')
vect.fit(corpus)
print(vect.get_feature_names())
```

```
['beautiful', 'blue', 'brown', 'dog', 'fox', 'jumps', 'lazy', 'love', 'quick', 'sky']
```

```
X = vect.transform(corpus)
print(X.shape)
print(X.toarray())
print(vect.inverse_transform(X)[0])
Doc_Term_Matrix = pd.DataFrame(X.toarray(),columns= vect.get_feature_names())
Doc_Term_Matrix
```

```
(3, 10)
[[2 2 0 0 0 0 0 1 0 2]
 [0 1 2 2 2 1 2 0 2 0]
 [0 1 2 2 2 0 2 0 2 0]]
['beautiful' 'blue' 'love' 'sky']
```

	beautiful	blue	brown	dog	fox	jumps	lazy	love	quick	sky
0	2	2	0	0	0	0	0	1	0	2
1	0	1	2	2	2	1	2	0	2	0
2	0	1	2	2	2	0	2	0	2	0

```
smatrix = vect.transform(corpus)# Sparse matrix representation. Useful for large collections
print(smatrix)
```

```
  (0, 0)        2
  (0, 1)        2
  (0, 7)        1
  (0, 9)        2
  (1, 1)        1
  (1, 2)        2
  (1, 3)        2
  (1, 4)        2
  (1, 5)        1
  (1, 6)        2
  (1, 8)        2
  (2, 1)        1
  (2, 2)        2
  (2, 3)        2
  (2, 4)        2
  (2, 6)        2
  (2, 8)        2
```

```
from sklearn.feature_extraction.text import TfidfTransformer
tfidf_transformer = TfidfTransformer()
X_tfidf = tfidf_transformer.fit_transform(X)
X_tfidf.shape
```

```
(3, 10)
```

```
Doc_tfidf_Matrix = pd.DataFrame(X_tfidf.toarray(),columns= vect.get_feature_names())
Doc_tfidf_Matrix
```

	beautiful	blue	brown	dog	fox	jumps	lazy	love	quick	sky
0	0.620314	0.366367	0.00000	0.00000	0.00000	0.000000	0.00000	0.310157	0.00000	0.620314
1	0.000000	0.164334	0.42322	0.42322	0.42322	0.278242	0.42322	0.000000	0.42322	0.000000
2	0.000000	0.171090	0.44062	0.44062	0.44062	0.000000	0.44062	0.000000	0.44062	0.000000

```
from sklearn.metrics.pairwise import cosine_similarity
similarity =cosine_similarity(X_tfidf,X_tfidf)
print(similarity)
```

```
[[1.         0.0602066  0.06268184]
 [0.0602066  1.         0.96051108]
 [0.06268184 0.96051108 1.        ]]
```

N-gram Word Features

- One issue with the bag of words representation is the loss of context. The BoW representation just focuses on words presence in isolation; it doesn't use the neighboring words to build a more meaningful representation.
- One way to overcome this issue is by allowing a vector representation using *N-grams* of words. In such a model, we use N successive words as features. Thus, in a bi-gram model, N = 2, two successive words will be used as features in the vector representations of documents. For N = 3, trigrams will be used to build the representation.
- The CountVectorizer from the sklearn library allows us to specify N.
- The N-gram representation does increase the document vector size.

Using Hashing to Build Document Vectors

- There are two main issues with the tf-idf representaion.
  - First, the vocabulary size can grow so much so as not to fit in the available memory for large corpus. In such a case, we need two passes over data.
  - The second issue arises in the context of online text classifiers such as spam classifiers for e-mails. When such a classifier encounters words not in its vocabulary, it ignores them. Spammers can take advantage of this by deliberately misspelling words in its message which when ignored by the spam filter will cause the spam message appear normal.

- Using hashing to build vectors overcomes the limitations mentioned above.

The following link provides more details on this topic.

https://iksinc.online/2019/12/21/numeric-representation-of-text-countvectorizer-to-hashingvectorizer/