

# Information Retrieval and Web Analytics (IT 3041)

- ▶ Lecturer in Charge - H. M. Samadhi Chathuranga Rathnayake

# Information Retrieval and Web Analytics (IT 3041)

## ► Course Assessment:

1. Assignment 1 (Practical Assignment + Lab Tasks) - 15 Marks + 5 Marks
2. Assignment 2 (Group Assignment) - 20 Marks
3. Mid Exam - 20 Marks
4. Final Exam - 40 Marks

## ► Course Textbook:

C. Manning, P. Raghavan, and H. Schütze, Introduction to Information Retrieval. Cambridge University Press, 2008. (available online at <https://nlp.stanford.edu/IR-book/information-retrieval-book.html>)

No	Topic	
1	Introduction to Information Retrieval Boolean models	
2	Term Vocabulary & Posting Lists	
3	Dictionaries and Tolerant Retrieval	
4	Scoring, term weighting and the vector space model	
5	Further enhancement to Scoring and results assembly and evaluation	
6	Mid Review	
7	Mid exam	
8	Web Crawling	
9	Personalization in IR Systems	
10	Introduction to Digital Marketing	
11	Text Classification	
12	Language Models for Information Retrieval	
13	Link Analysis in IR Systems	
14	Final Exam	

# Introduction to Information Retrieval

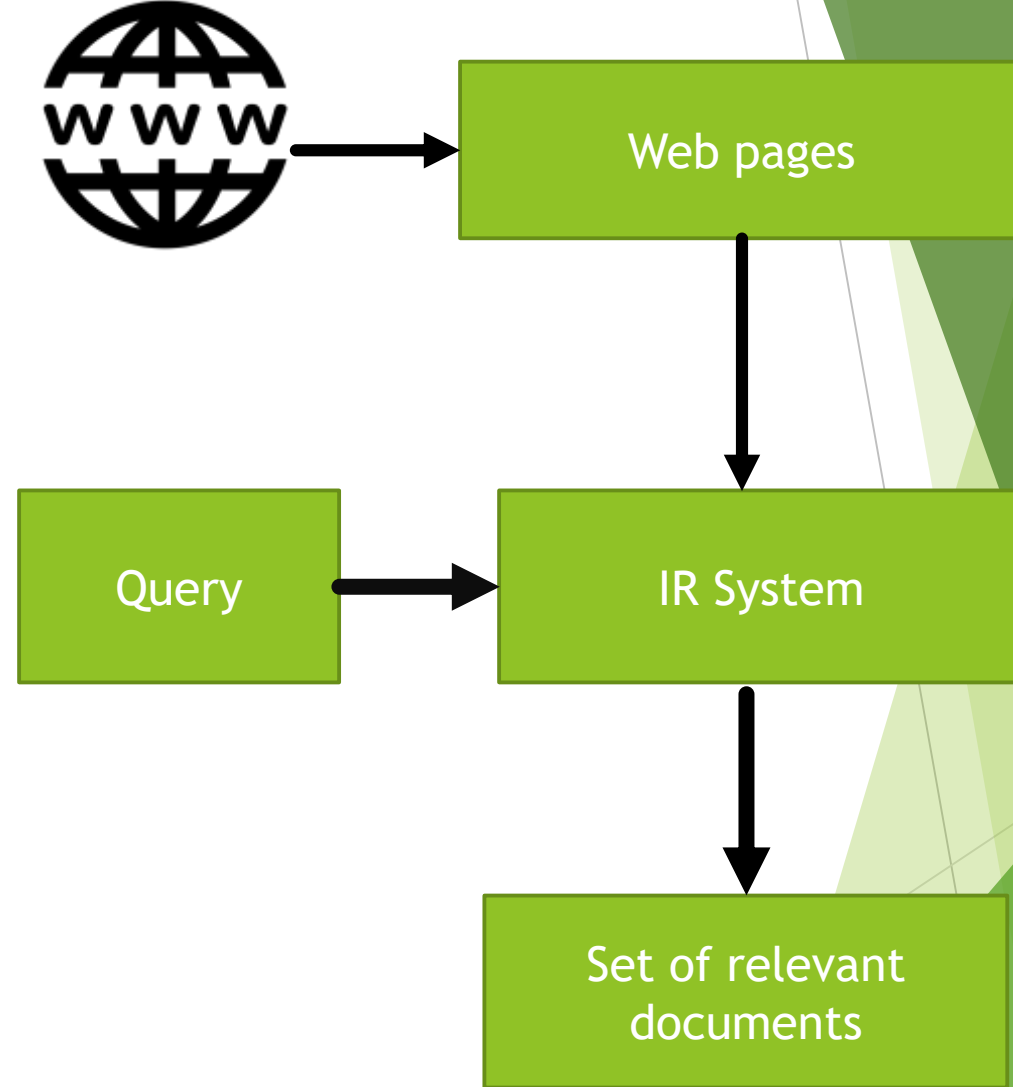
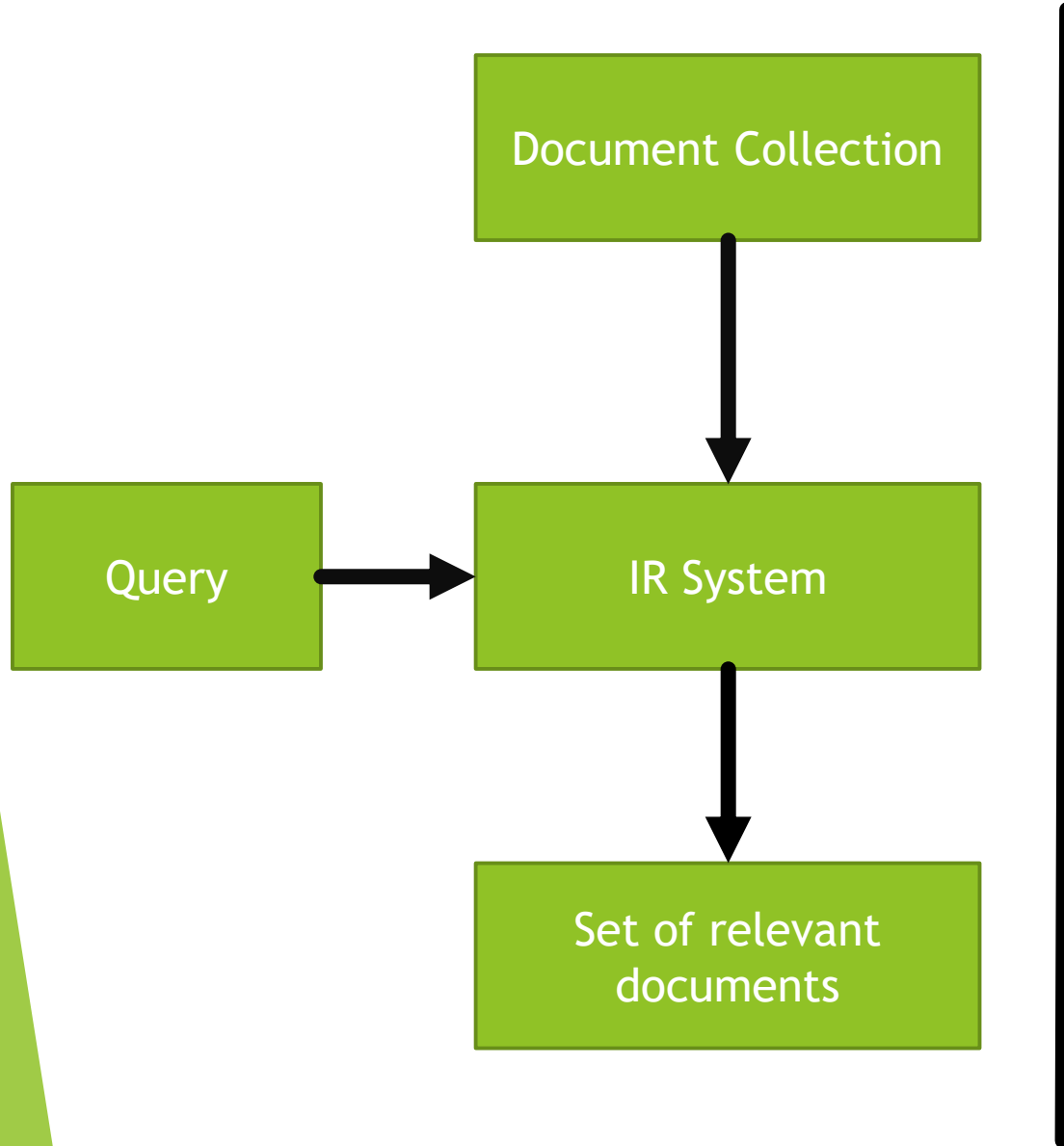
# Information Retrieval

- Manning et al, 2008:

Information Retrieval (IR) is finding material (usually documents) of an unstructured nature (usually text) that satisfies an information need from within large collections (usually stored on computers).

- These days we frequently think first of web search, but there are many other cases:
  - E-mail search
  - Searching your laptop
  - Corporate knowledge bases
  - Legal information retrieval

# IR Basics



# IR vs. databases:

## Structured vs unstructured data

- 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 data

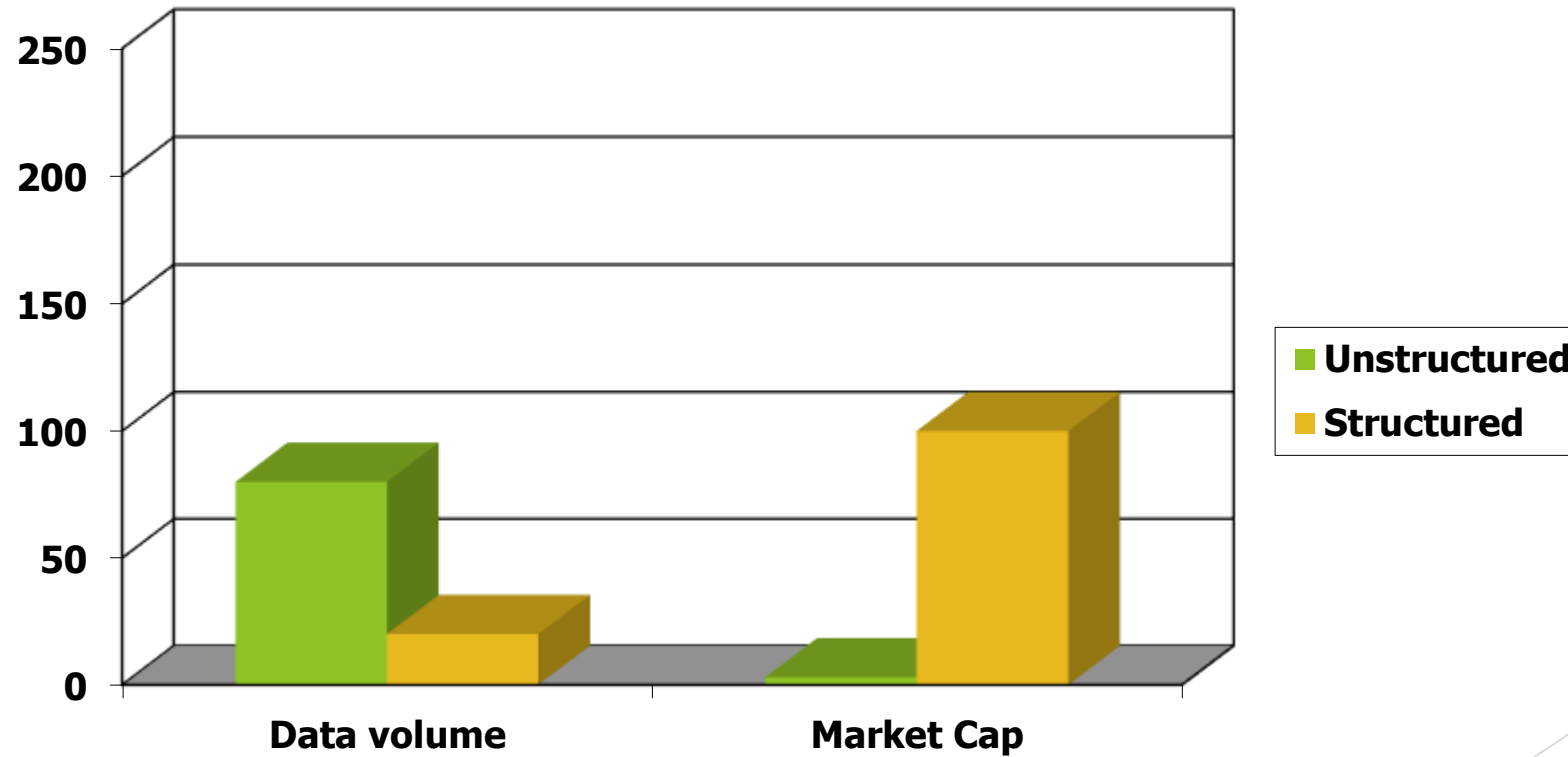
- Typically refers to free text
- No data model
- Data available in naïve format
- Allows
  - Keyword queries including operators
  - More sophisticated “concept” queries e.g.,
    - find all web pages dealing with *drug abuse*
- Classic model for searching text documents



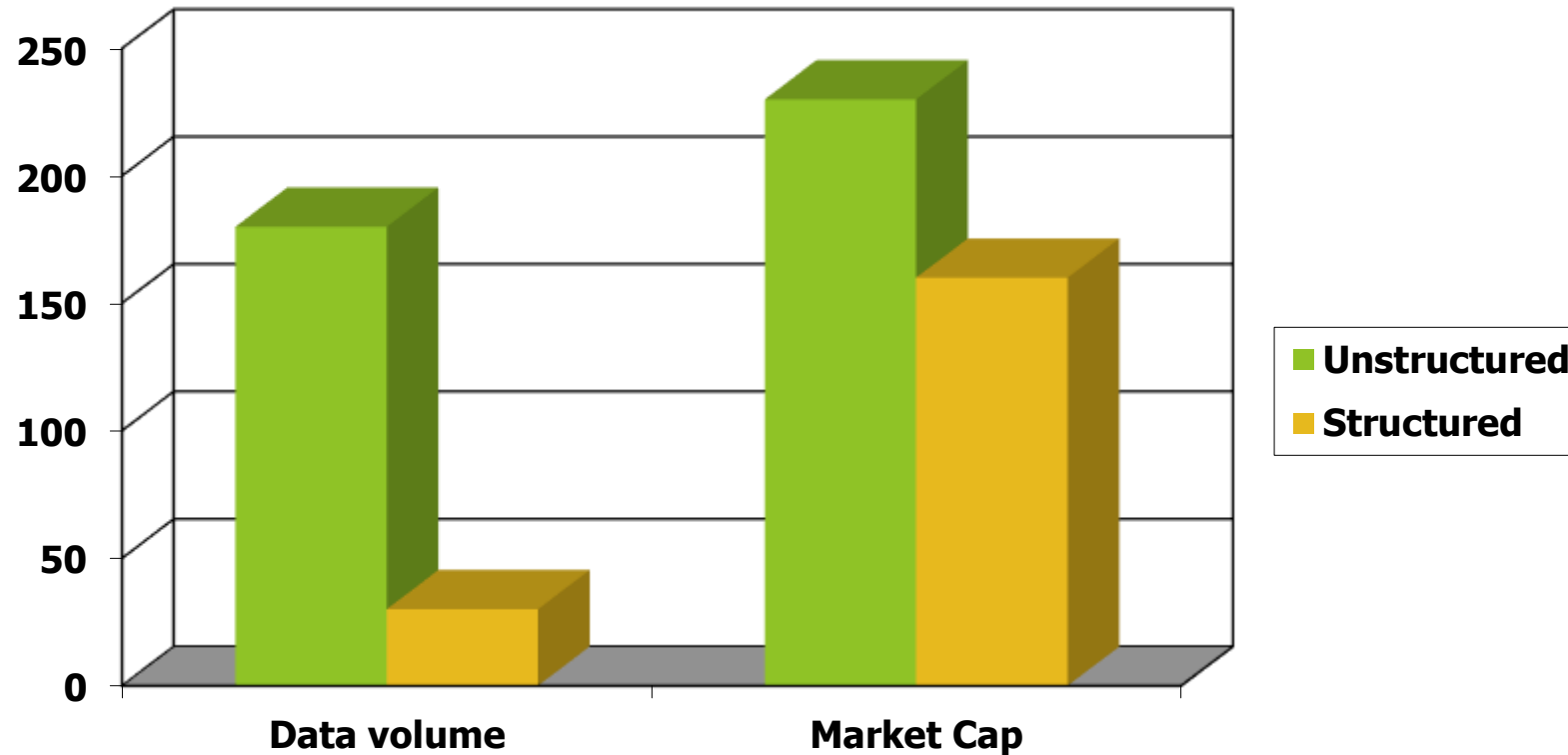
# Semi-structured data

- Type of structured data that does not fit into the formal structure of a relational database.
- In fact almost no data is “unstructured”
- E.g., this slide has distinctly identified zones such as the *Title* and *Bullets*
  - ... to say nothing of linguistic structure
- Facilitates “semi-structured” search such as
  - *Title* contains data AND *Bullets* contain search
- Or even
  - *Title* is about Object Oriented Programming AND *Author* something like stro\*rup
  - where \* is the wild-card operator

# Unstructured (text) vs. structured (database) data in the mid-nineties



# Unstructured (text) vs. structured (database) data today



# Information Need and Relevance

- An information need is the topic about which the user desires to know more about.
- A query is what the user conveys to the computer in an attempt to communicate the information need.
- A document is relevant if the user perceives that it contains information of value with respect to their personal information need.

## Type of Information Needs

- Known-item search
- Precise information seeking search
- Open-ended search (“topical search”)

## Information Scarcity & Abundance Problem

- Information scarcity problem (or needle-in-haystack problem): hard to find rare information
- Information abundance problem (for more clear-cut information needs): redundancy of obvious information

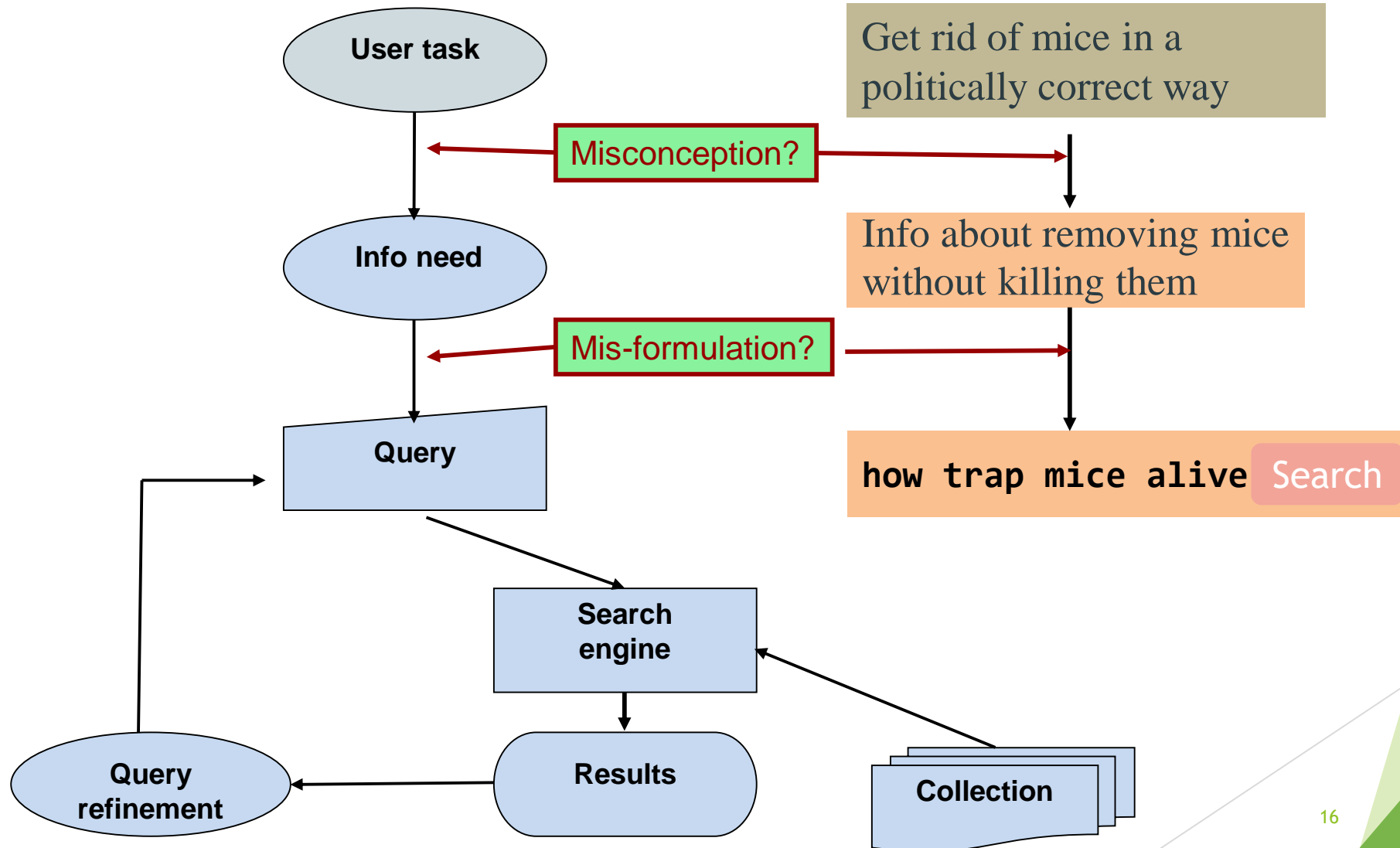
## Information Relevance

- Are the retrieved documents
  - about the target subject up-to-date?
  - from a trusted source?
  - satisfying the user's needs?
- How should we rank documents in terms of these factors?

# Basic assumptions of Information Retrieval

- **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**

# The classic search model

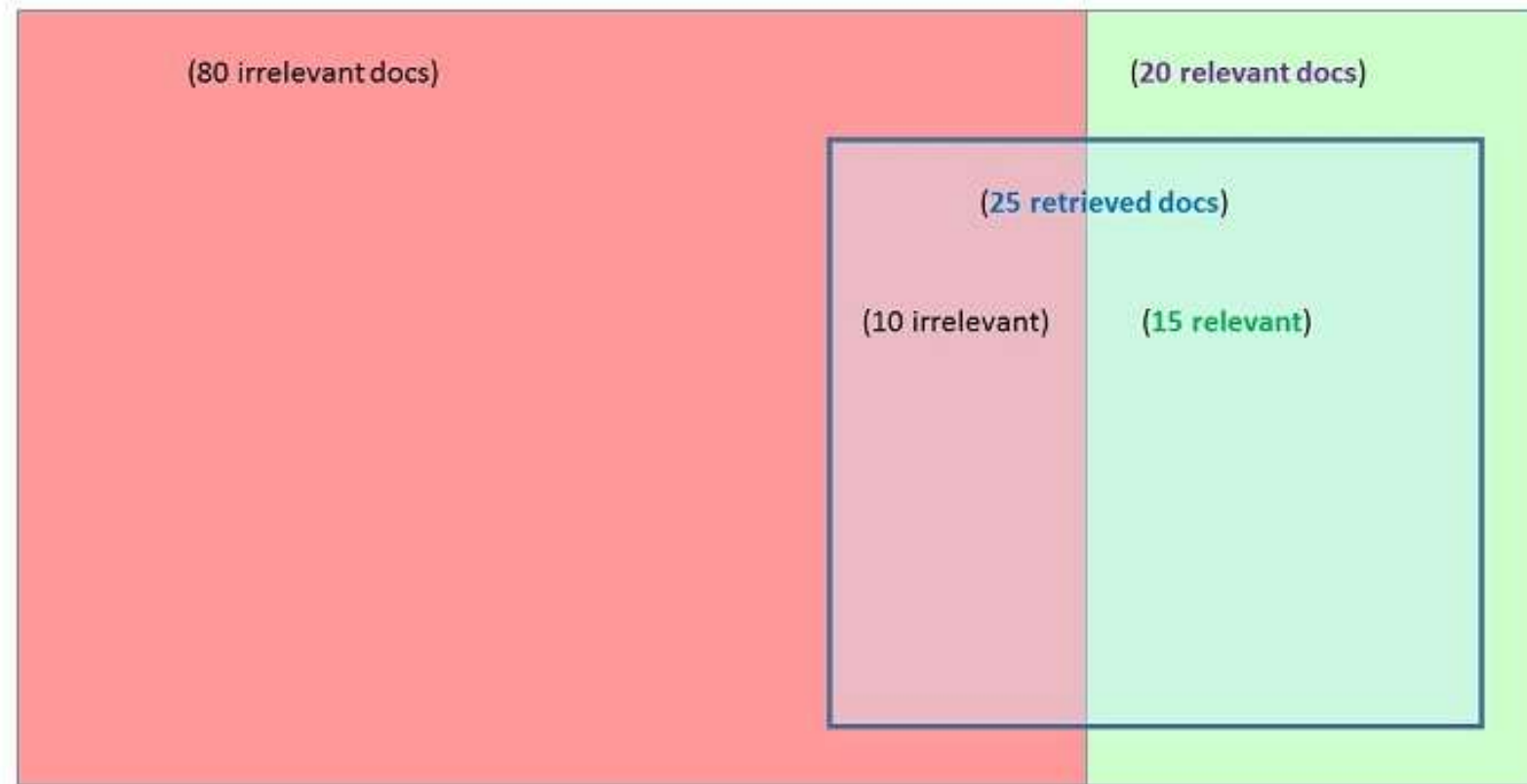




# How good are the retrieved docs?

- The effectiveness of an IR system (i.e., the quality of its search results) is determined by two key statistics about the system's returned results for a query:
- **Precision**: What fraction of the returned results are relevant to the information need?
  - $\text{Total number of documents retrieved that are relevant} / \text{Total number of documents that are retrieved}$ .
- **Recall**: What fraction of the relevant documents in the collection were returned by the system?
  - $\text{Total number of documents retrieved that are relevant} / \text{Total number of relevant documents in the database}$ .
- What is the best balance between the two?
  - Easy to get perfect recall: just retrieve everything
  - Easy to get good precision: retrieve only the most relevant

# How good are the retrieved docs?



# Information Retrieval Today

- Web search (Google, Bing, Yahoo )
  - Search ground are billions of documents on millions of computers
  - issues: spidering; efficient indexing and search; malicious manipulation to boost search engine rankings
- Enterprise and institutional search (PubMed, LexisNexis )
  - e.g company's documentation, patents, research articles
  - often domain-specific
  - Centralised storage; dedicated machines for search.
  - Most prevalent IR evaluation scenario: US intelligence analyst's searches
- Personal information retrieval (email, pers. documents; )
  - e.g., Mac OS X Spotlight; Windows' Instant Search
  - Issues: different file types; maintenance-free, lightweight to run in background

# Boolean Model

# Unstructured data in 1620

- 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 lines containing *Calpurnia*?
- Why is that not the answer?
  - Slow (for large corpora)
  - NOT *Calpurnia* is non-trivial
  - Other operations (e.g., find the word *Romans* near *countrymen*) not feasible
  - Ranked retrieval (best documents to return)
    - Later lectures

# Term-document incidence matrices

	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.
- To answer query: compute the results for our query using bitwise AND between
- vectors for **Brutus**, **Caesar** and complement (**Calpurnia**)
  - $110100 \text{ AND}$
  - $110111 \text{ AND}$
  - $101111 =$
  - $100100$

	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.



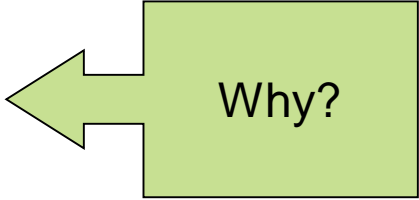


# Bigger collections

- 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 Term-Document incidence 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(lots of zeros).
- What's a better representation?
  - We only record the 1 positions.



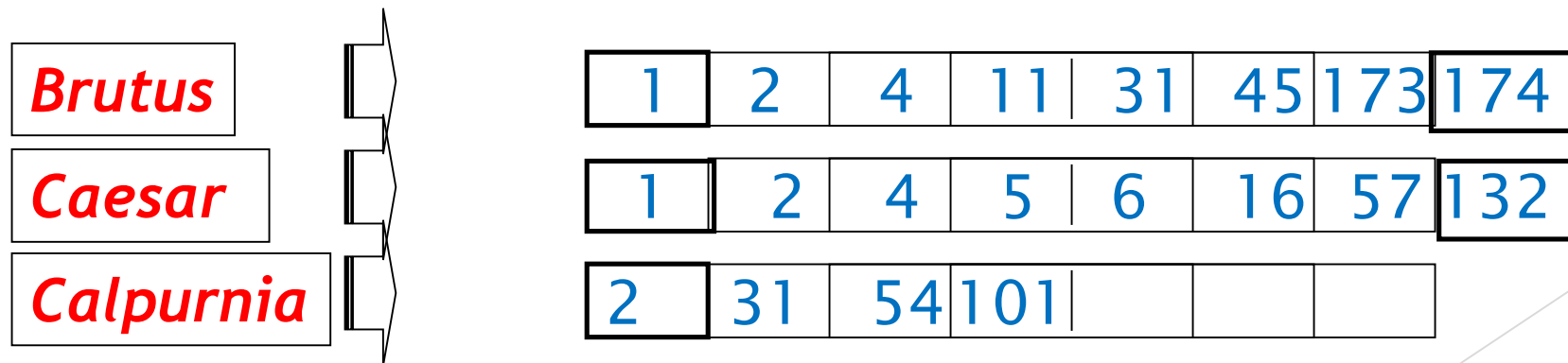
Why?

# Can't build the Term-Document incidence matrix

- Observation: the term-document matrix is very sparse
- Contains no more than one billion 1s.
- Better representation: only represent the things that do occur
- Term-document matrix has other disadvantages, such as lack of support for more complex query operators (e.g., proximity search)
- We will move towards richer representations, beginning with the **inverted index**.

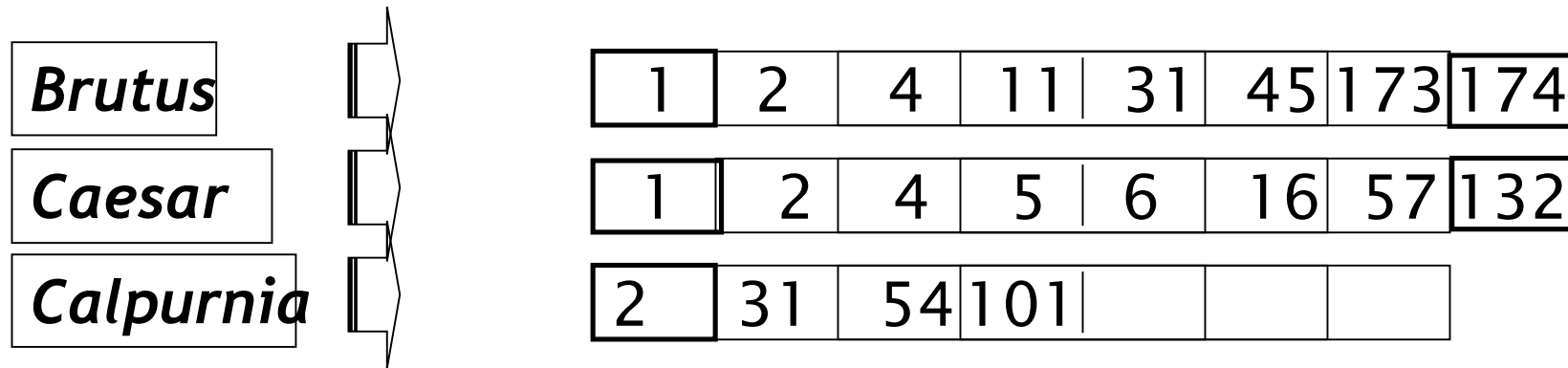
# Inverted index

- The inverted index consists of
  - a **dictionary** of terms (also: lexicon, vocabulary)
  - and a **postings list** for each term, i.e., a list that records which documents the term occurs in.



# Inverted index

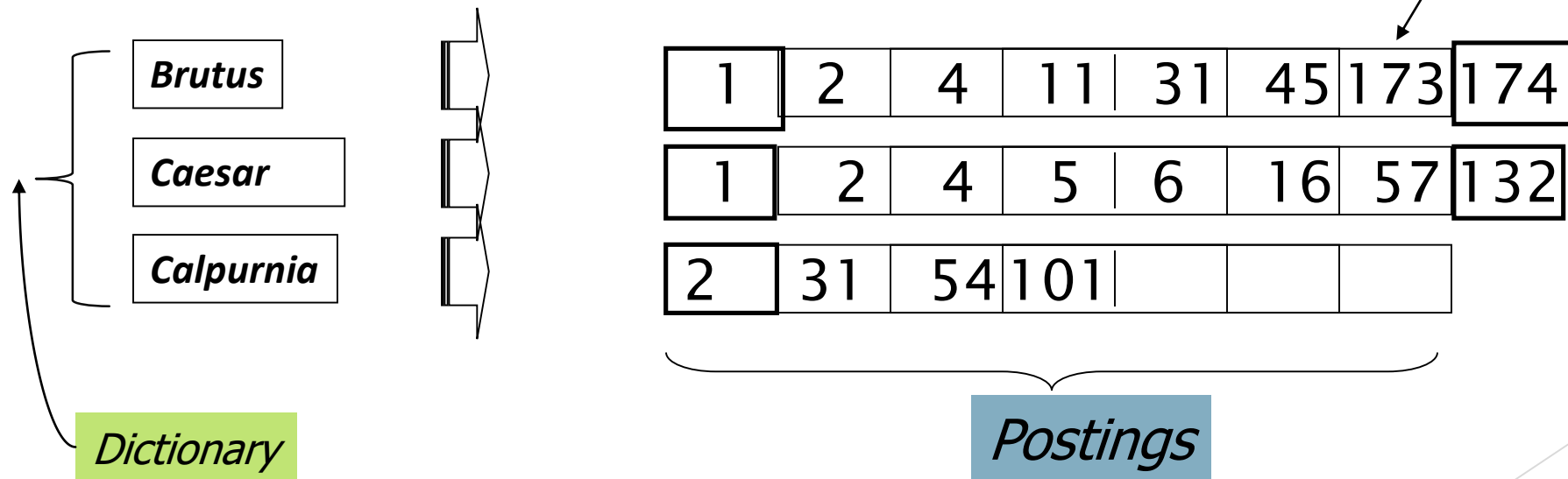
- For each term  $t$ , we must store a list of all documents that contain  $t$ .
  - Identify each doc by a **docID**, a document serial number
- Can we use fixed-size arrays for this?



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

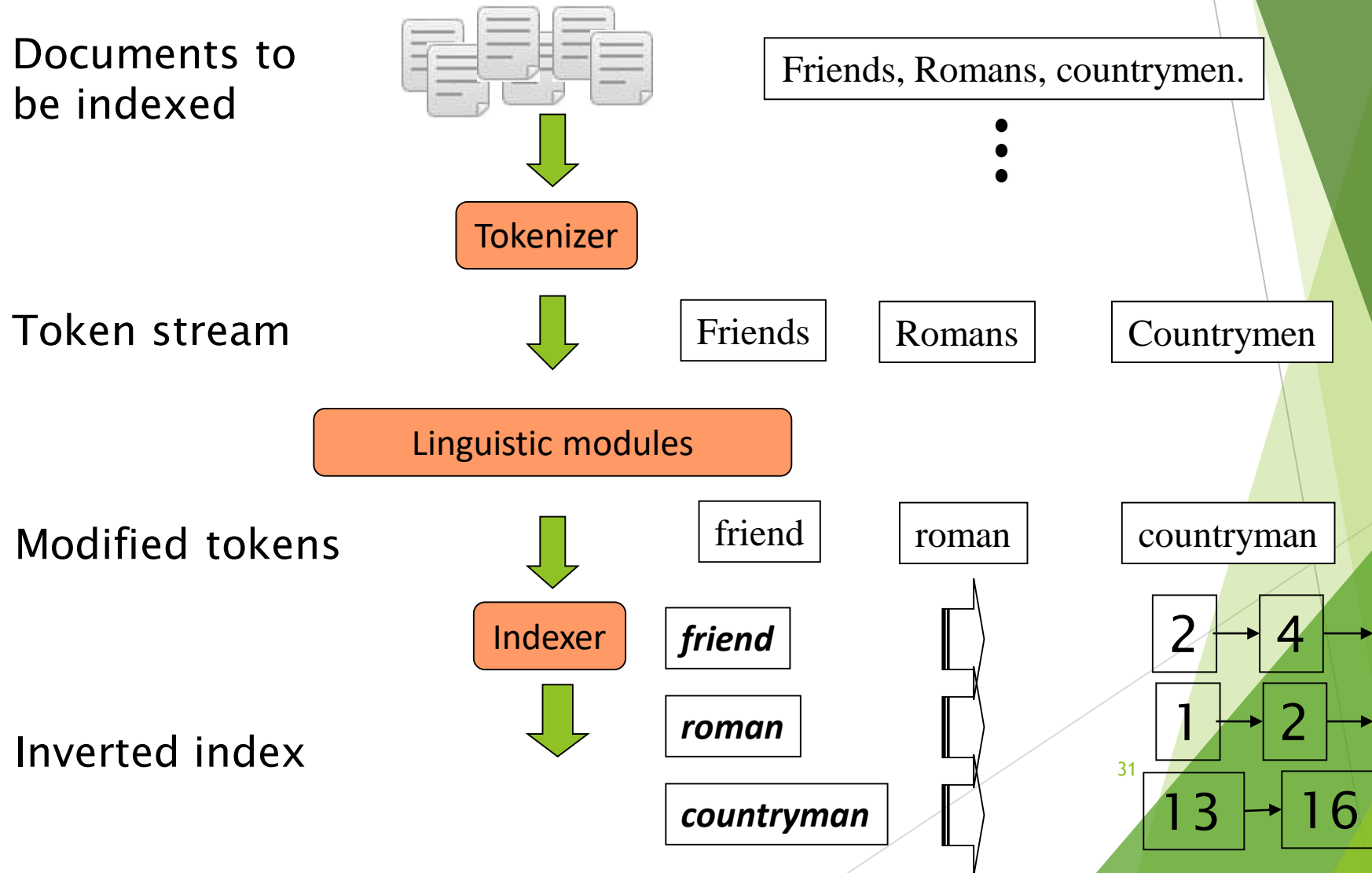
# Inverted index

- 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 arrays
    - Some tradeoffs in size/ease of insertion



Sorted by docID (more later on why).

# 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
33	

# Indexer steps: Sort

- Sort by terms
  - And then docID



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



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

# Indexer steps: Dictionary & Postings

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

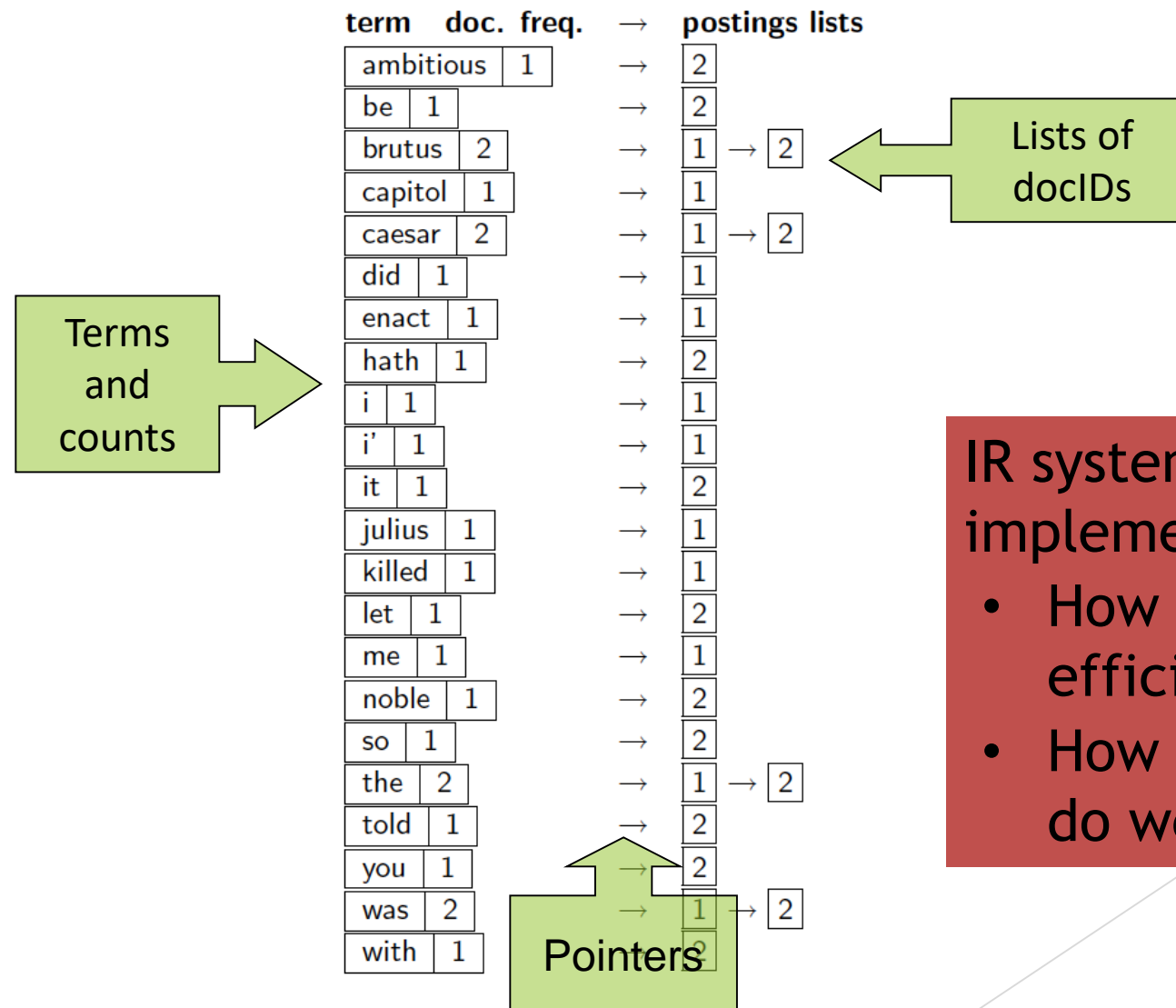
Why frequency?  
Will discuss later.

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

# Where do we pay in storage?

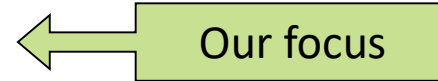


## IR system implementation

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

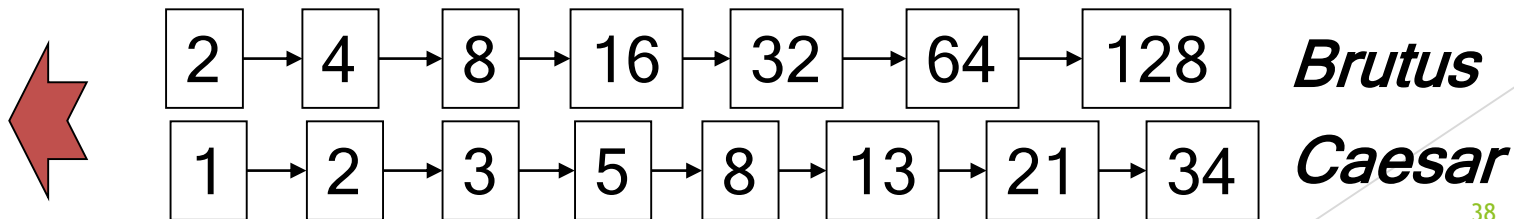
# The index we just built

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



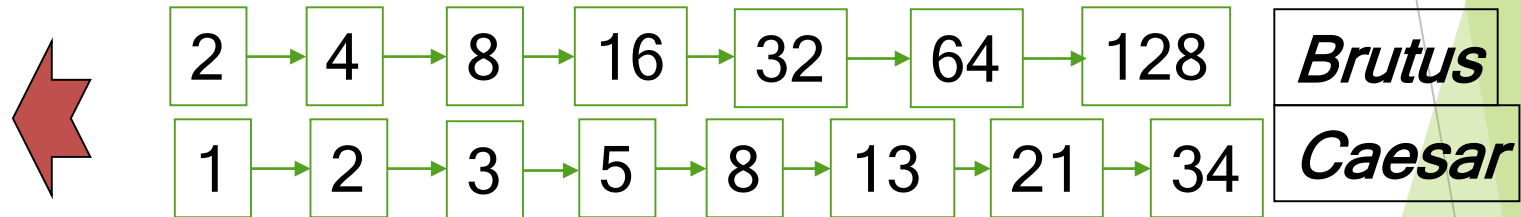
# Query processing: AND

- 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 document sets):



# 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.

# Intersecting two postings lists (a “merge” algorithm)

```
INTERSECT( $p_1, p_2$ )
1   $answer \leftarrow \langle \rangle$ 
2  while  $p_1 \neq \text{NIL}$  and  $p_2 \neq \text{NIL}$ 
3  do if  $docID(p_1) = docID(p_2)$ 
4      then  $\text{ADD}(answer, docID(p_1))$ 
5           $p_1 \leftarrow next(p_1)$ 
6           $p_2 \leftarrow next(p_2)$ 
7      else if  $docID(p_1) < docID(p_2)$ 
8          then  $p_1 \leftarrow next(p_1)$ 
9          else  $p_2 \leftarrow next(p_2)$ 
10 return  $answer$ 
```

Brutus 1 2 4 11 31 45 173 174  
Calpurnia 2 31 54 101  
Intersection 2 31



# Boolean queries: Exact match

- The **Boolean retrieval model** is being able to ask a query that is a Boolean expression:
  - Boolean Queries are queries using *AND*, *OR* and *NOT* to join query terms
    - Views each document as a set of words
    - Is precise: document matches condition or not.
  - Perhaps the simplest model to build an IR system on
- Primary commercial retrieval tool for 3 decades.
- Many search systems you still use are Boolean:
  - Email, library catalog, Mac OS X Spotlight

# Boolean queries: More general merges

- Exercise: Adapt the merge for the queries:
- *Brutus AND NOT Caesar*
- *Brutus OR NOT Caesar*
- Can we still run through the merge in time  $O(x+y)$ ? What can we achieve?

# Merging

What about an arbitrary Boolean formula?

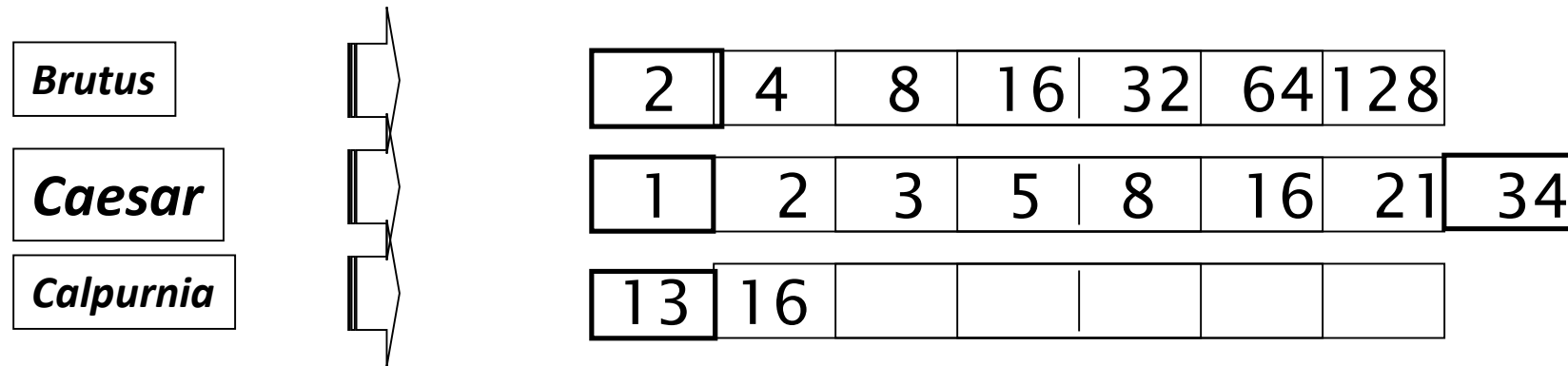
*Brutus AND Calpurnia AND Caesar*

Can we always merge in “linear” time?

- ▶ Linear in what?
- ▶ Can we do better?

# Query optimization

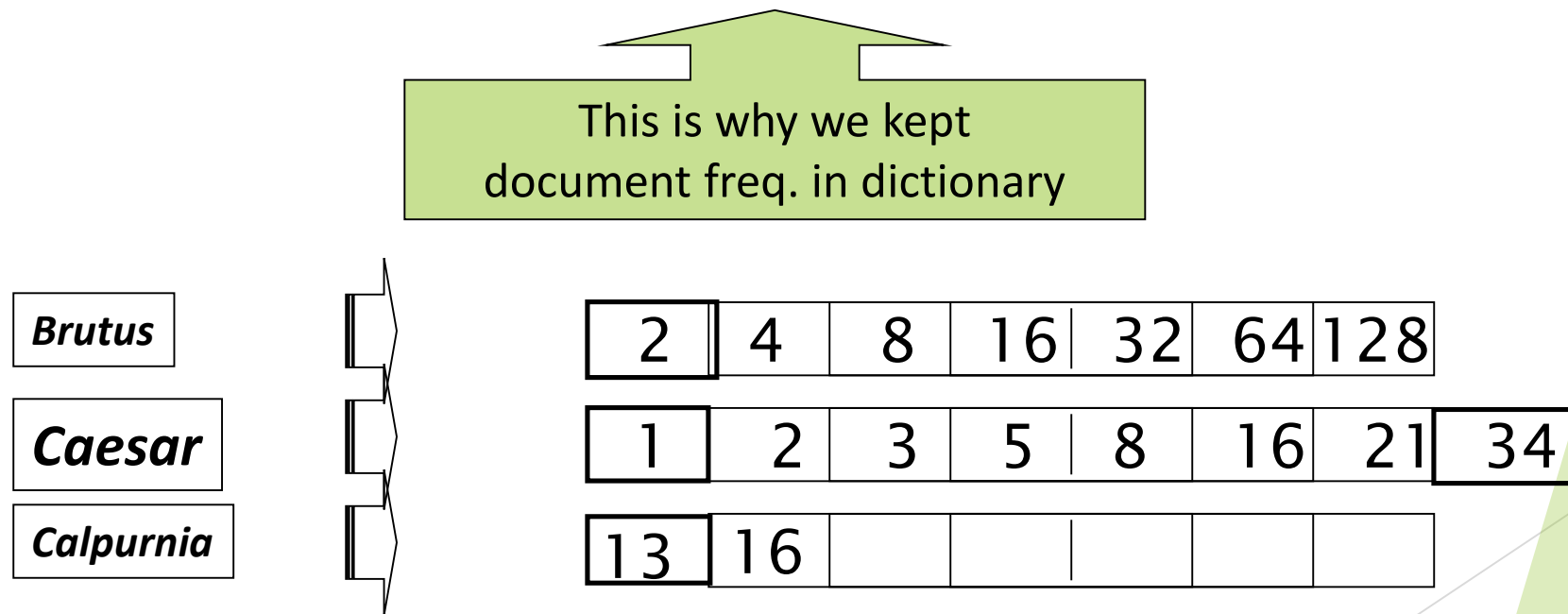
- What is the best order for query processing?
- Consider a query that is an *AND* of  $n$  terms.
- For each of the  $n$  terms, get its postings, then *AND* (intersect) them together.



**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.

# Exercise

- Recommend a query processing order for

*(tangerine OR trees) AND  
(marmalade OR skies) AND  
(kaleidoscope OR eyes)*

- Which two terms should we process first?

Term	Freq
eyes	213312
kaleidoscope	87009
marmalade	107913
skies	271658
tangerine	46653
trees	316812

# Query processing exercises

- **Exercise:** If the query is *friends AND romans AND (NOT countrymen)*, how could we use the freq of *countrymen*?
- **Exercise:** Extend the merge to an arbitrary Boolean query. Can we always guarantee execution in time linear in the total postings size?
- **Hint:** Begin with the case of a Boolean *formula* query: in this, each query term appears only once in the query.



# Exercise

- ▶ Try the search feature at <http://www.rhymezone.com/shakespeare/>
- ▶ Write down five search features you think it could do better