

# Introduction to **Information Retrieval**

# Book

---

- Introduction to Information Retrieval by C. Manning et al

# Information Retrieval

---

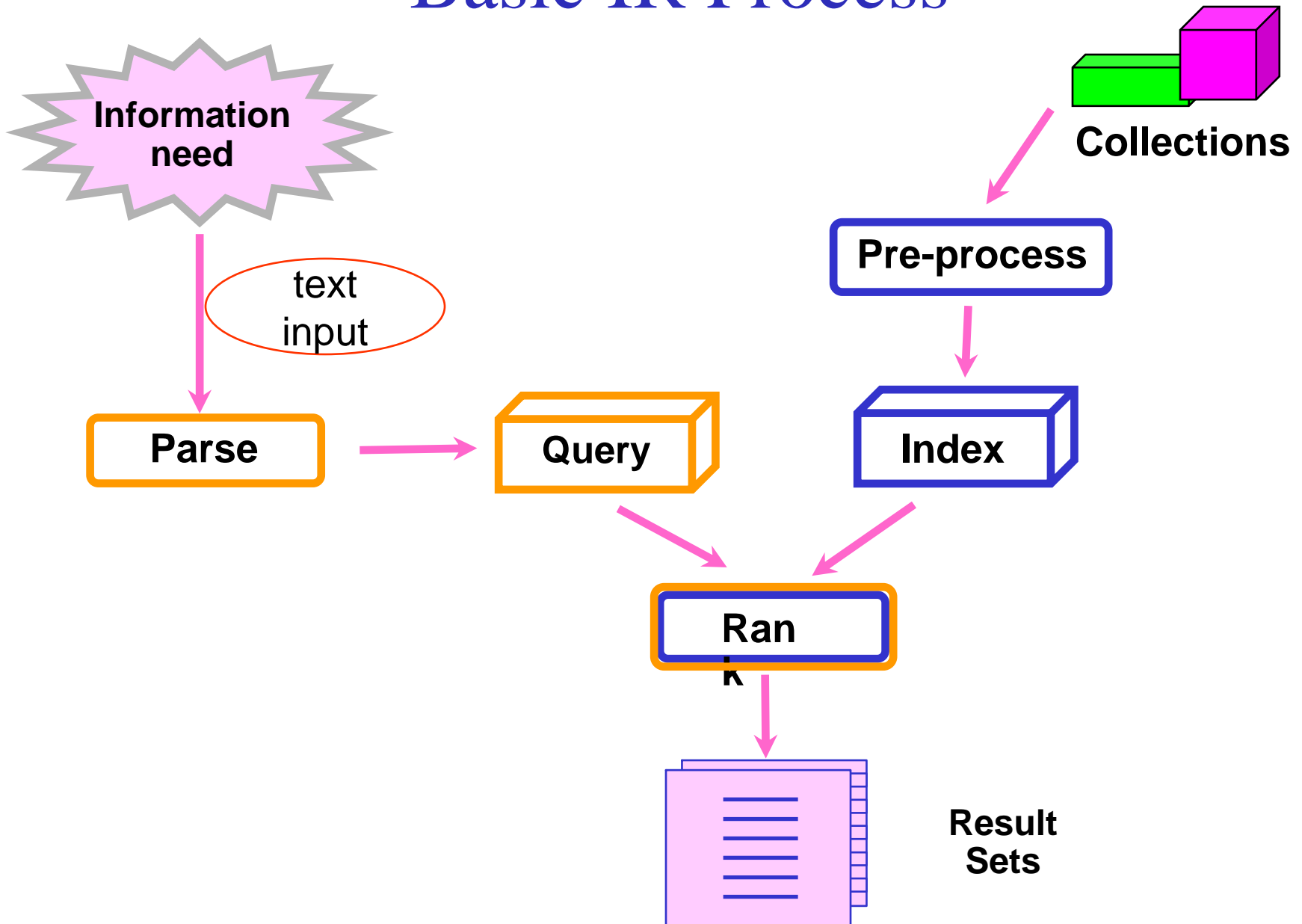
- Information Retrieval (IR) is finding material (usually documents) of an unstructured nature (usually text) that satisfies an information need from within large collections (usually stored on computers).
- These days we frequently think first of web search, but there are many other cases:
  - E-mail search
  - Searching your laptop
  - Corporate knowledge bases
  - Legal information retrieval

# 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**

# Basic IR Process



# How good are the retrieved docs?

---

- *Precision* : Fraction of retrieved docs that are relevant to the user's **information need**
- *Recall* : Fraction of relevant docs in collection that are retrieved
  - More precise definitions and measurements to follow later

# Example retrieval

---

- 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: take the vectors for ***Brutus, Caesar*** and ***Calpurnia*** (complemented)  $\boxtimes$  bitwise *AND*.
  - 110100 *AND*
  - 110111 *AND*
  - 101111 =
  - **100100**

	Antony and Cleopatra	Julius Caesar	The Tempest	Hamlet	Othello	Macbeth
Antony	1	1	0	0	0	1
Brutus	1	1	0	1	0	0
Caesar	1	1	0	1	1	1
Calpurnia	0	1	0	0	0	0
Cleopatra	1	0	0	0	0	0
mercy	1	0	1	1	1	1
worser	1	0	1	1	1	0

# Answers to query

---

- Antony and Cleopatra, Act III, Scene ii

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

- Hamlet, Act III, Scene ii

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



# Bigger collections

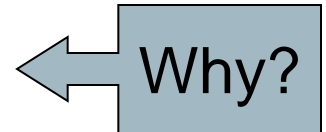
---

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

# Can't build the matrix

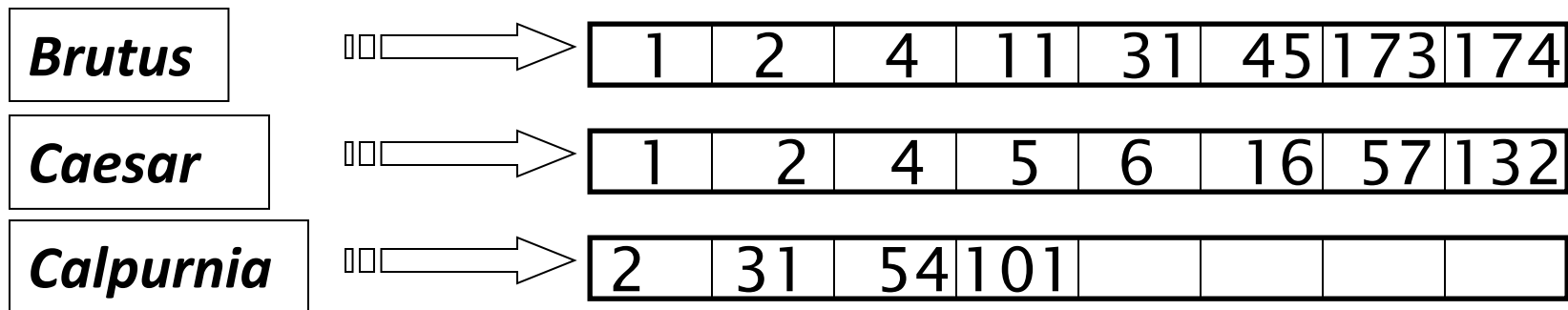
---

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



# Inverted index

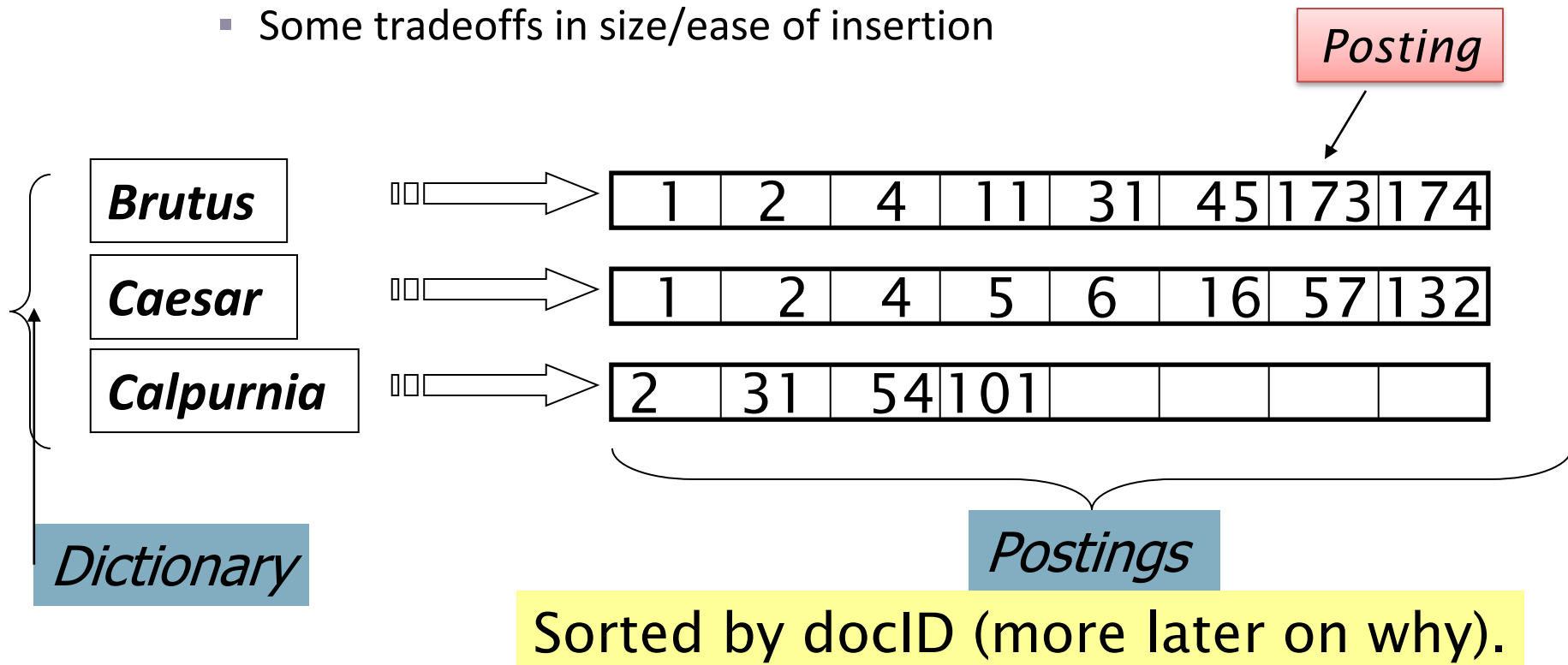
- For each term  $t$ , we must store a list of all documents that contain  $t$ .
  - Identify each doc by a **docID**, a document serial number
- Can we use fixed-size arrays for this?



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



# Inverted index construction

Collect  
Documents to  
be indexed



Friends, Romans, countrymen.  
⋮

Tokenizer

Token stream

Friends

Romans

Countrymen

Linguistic  
preprocessing

Modified tokens

friend

roman

countryman

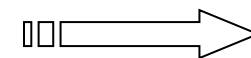
Indexer

Inverted  
index

**friend**

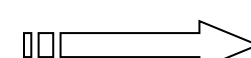
**roman**

**countryman**



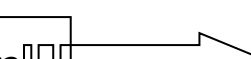
2

4



1

2



13

16  
23

# Initial stages of text processing

---

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



# Indexer steps: Token sequence

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

Doc 1

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

Doc 2

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



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

# Indexer steps: Sort

- Sort by terms
  - At least conceptually
    - 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

# Indexer steps: Dictionary & Postings

- Multiple term entries in a single document are merged.
- Instances of same term grouped: split into Dictionary and Postings
- Doc. frequency information is added.

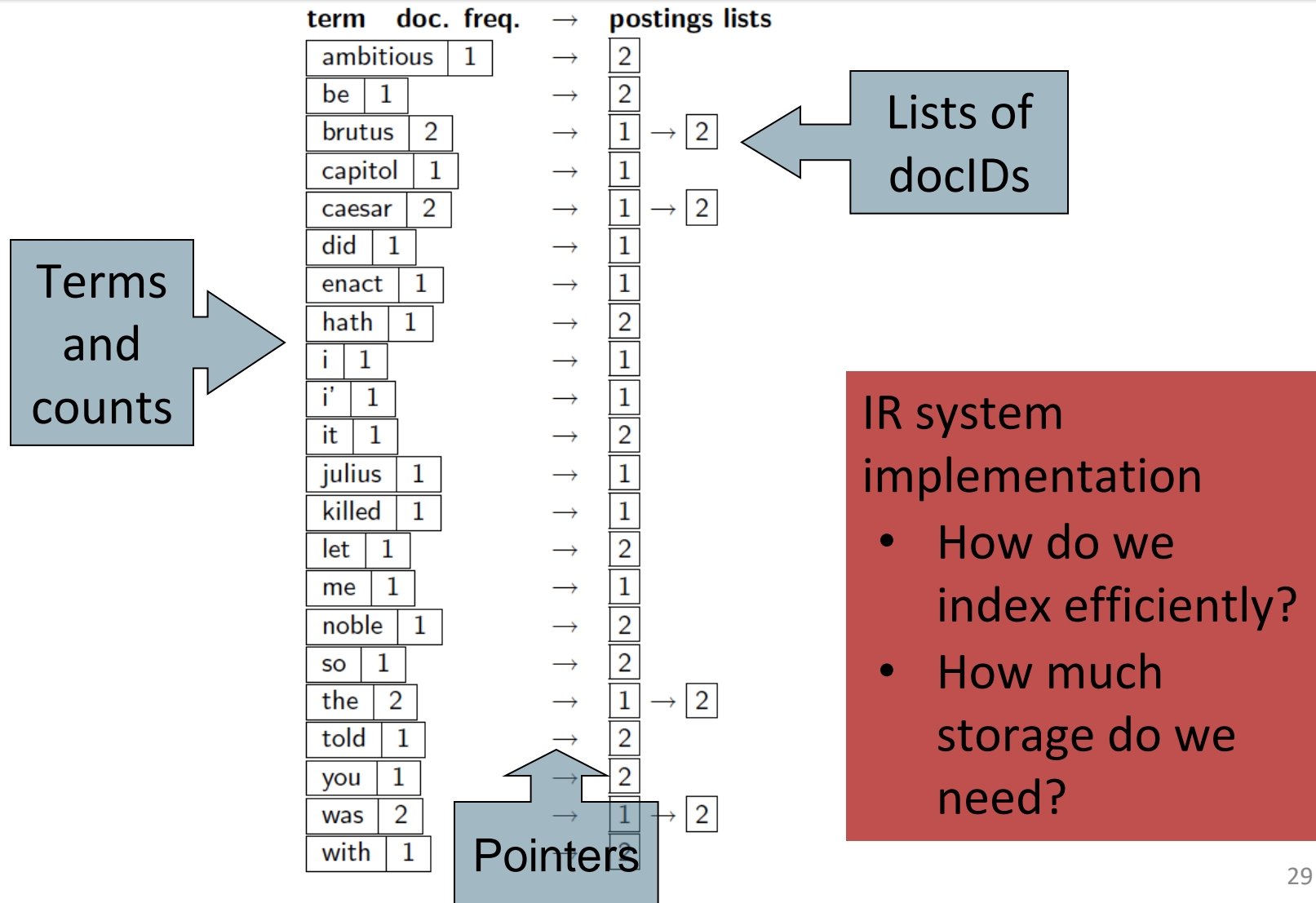
Why frequency?  
Will discuss later.

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?



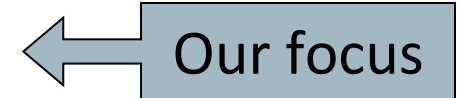
# Introduction to **Information Retrieval**

Query processing with an inverted index

# The index we just built

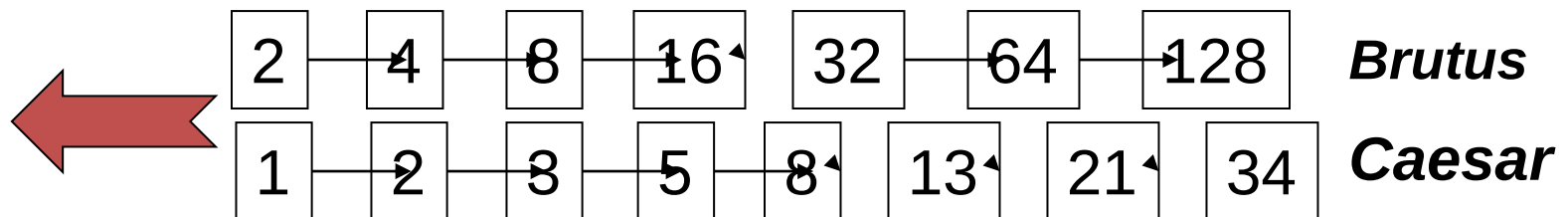
---

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



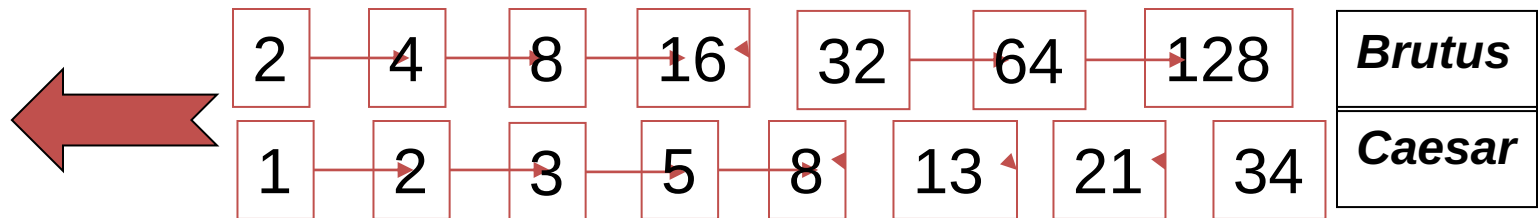
# Query processing: AND

- 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  $answer$ 
```

# Introduction to **Information Retrieval**

The Boolean Retrieval Model  
& Extended Boolean Models

# 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.

# Boolean queries:

## More general merges

---

- Exercise: Adapt the merge for the queries:

***Brutus AND NOT Caesar***

***Brutus OR NOT Caesar***

# Merging

---

What about an arbitrary Boolean formula?

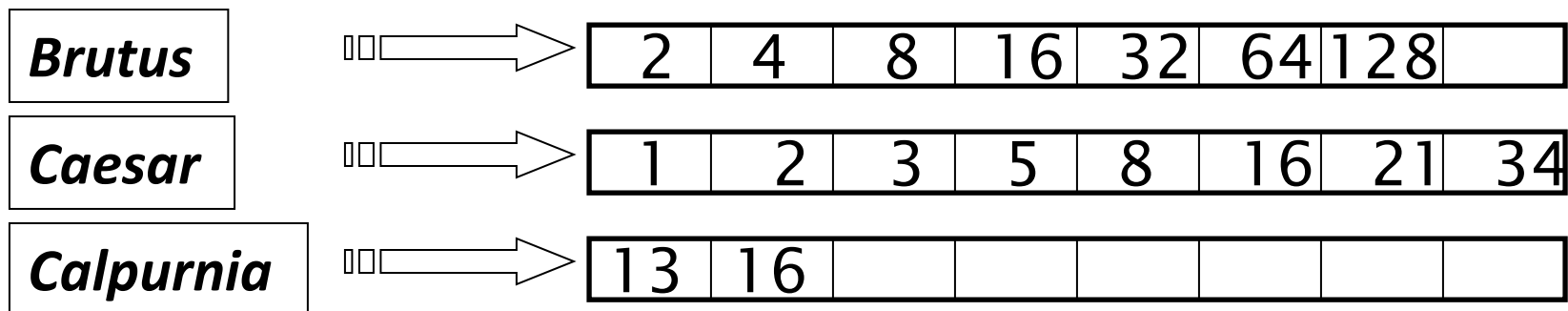
*(Brutus OR Caesar) AND NOT*

*(Antony OR Cleopatra)*

- Can we always merge in “linear” time?

# 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* 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.*

This is why we kept  
document freq. in dictionary

<b>Brutus</b>	⇒	2	4	8	16	32	64	128	
<b>Caesar</b>	⇒	1	2	3	5	8	16	21	34
<b>Calpurnia</b>	⇒	13	16						

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.

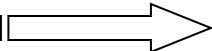
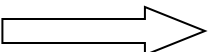


# Introduction to **Information Retrieval**

Phrase queries and positional indexes

# Phrase queries

---

- We want to be able to answer queries such as ***“stanford university”*** – as a phrase
- Thus the sentence *“I went to university at Stanford”* is not a match.
  - stanford □ □ 
  - university □ □ 
- For this, it no longer suffices to store only *<term : docs>* entries

DOES NOT ENSURE  
TOGETHER!

# A first attempt: Biword indexes

---

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

# Longer phrase queries

---

- Longer phrases can be processed by breaking them down
- ***stanford university palo alto*** can be broken into the Boolean query on biwords:

***stanford university AND university palo AND palo alto***

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



Can have false positives!

# Issues for biword indexes

---

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

## Solution 2: Positional indexes

---

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

<***term***, number of docs containing ***term***;

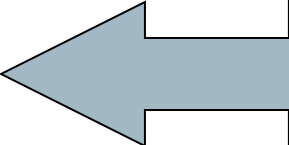
*doc1*: position1, position2 ... ;

*doc2*: position1, position2 ... ;

etc.>

# Positional index example

*<be*: 993427;  
*1*: 7, 18, 33, 72, 86, 231;  
*2*: 3, 149;  
*4*: 17, 191, 291, 430, 434;  
*5*: 363, 367, ...>



Which of docs  
*1,2,4,5*  
could contain “*to be*  
*or not to be*”?

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

# Processing a phrase query

---

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



# Positional index size

---

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