

Introduction to **Information Retrieval**

Boolean Retrieval Model

A First Information Retrieval Example

- **Ad hoc retrieval**: Find documents in a collection of documents (corpus), relevant to a certain user need
- **Boolean retrieval model**: Model in which queries are posed as Boolean expressions
- Example: Shakespeare
 - Find all Shakespeare plays that contain the words
BRUTUS AND CAESAR AND NOT CALPURNIA

– BRUTE Force Approach

- One could *grep* all of Shakespeare's plays for BRUTUS and CAESAR, then strip out plays containing CALPURNIA
 - Unix command grep(parse grep) on documents, linear search on documents.
- Why is that not the answer?
 - Slow (for large corpora=collection) -> performance
 - NOT Calpurnia is non-trivial
 - Other operations (e.g., find the word ROMANS NEAR COUNTRYMEN) not feasible
 - Ranked retrieval (best documents to return)

2. 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) → 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

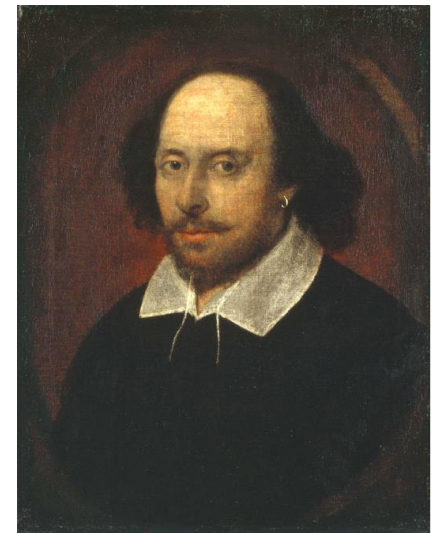
Answers to query

- 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

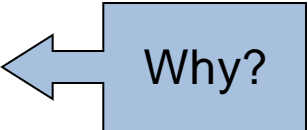
Lord Polonius: I did enact Julius **Caesar** I was killed i' the
Capitol; **Brutus** killed me.



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.

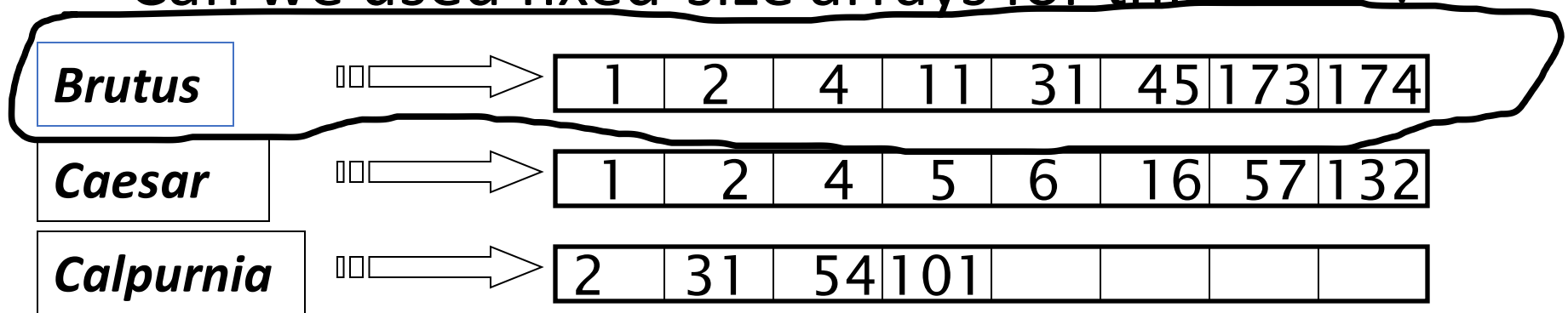
Introduction to **Information Retrieval**

The Inverted Index

The key data structure underlying
modern IR

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?

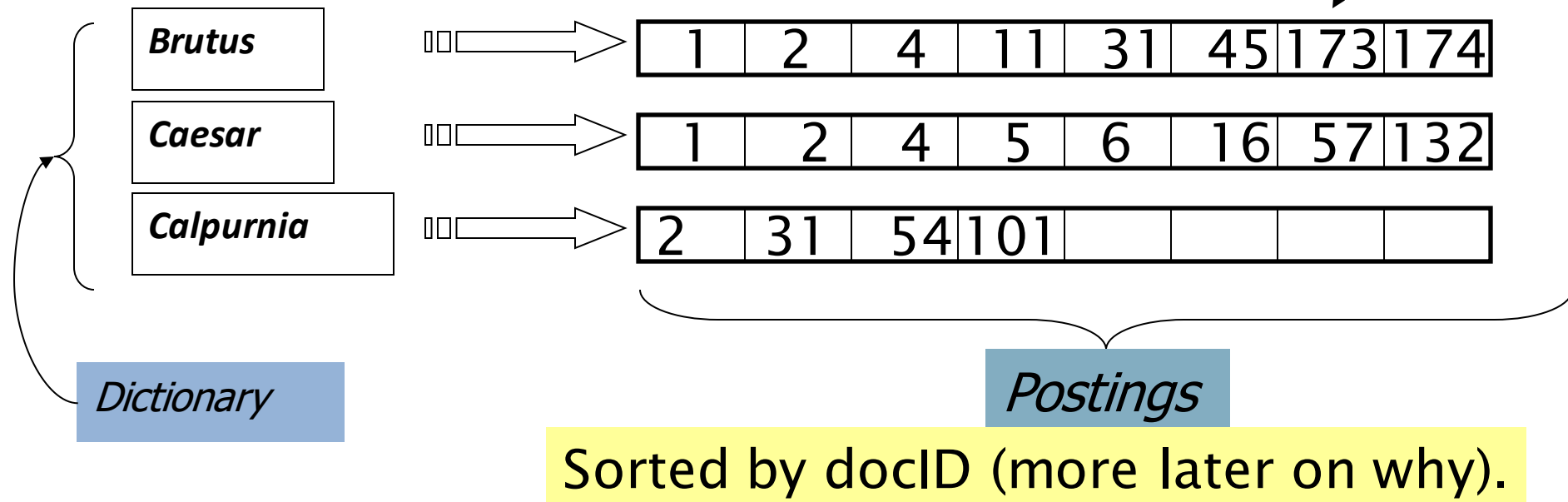


What happens if the word **Caesar** is added to document 14?

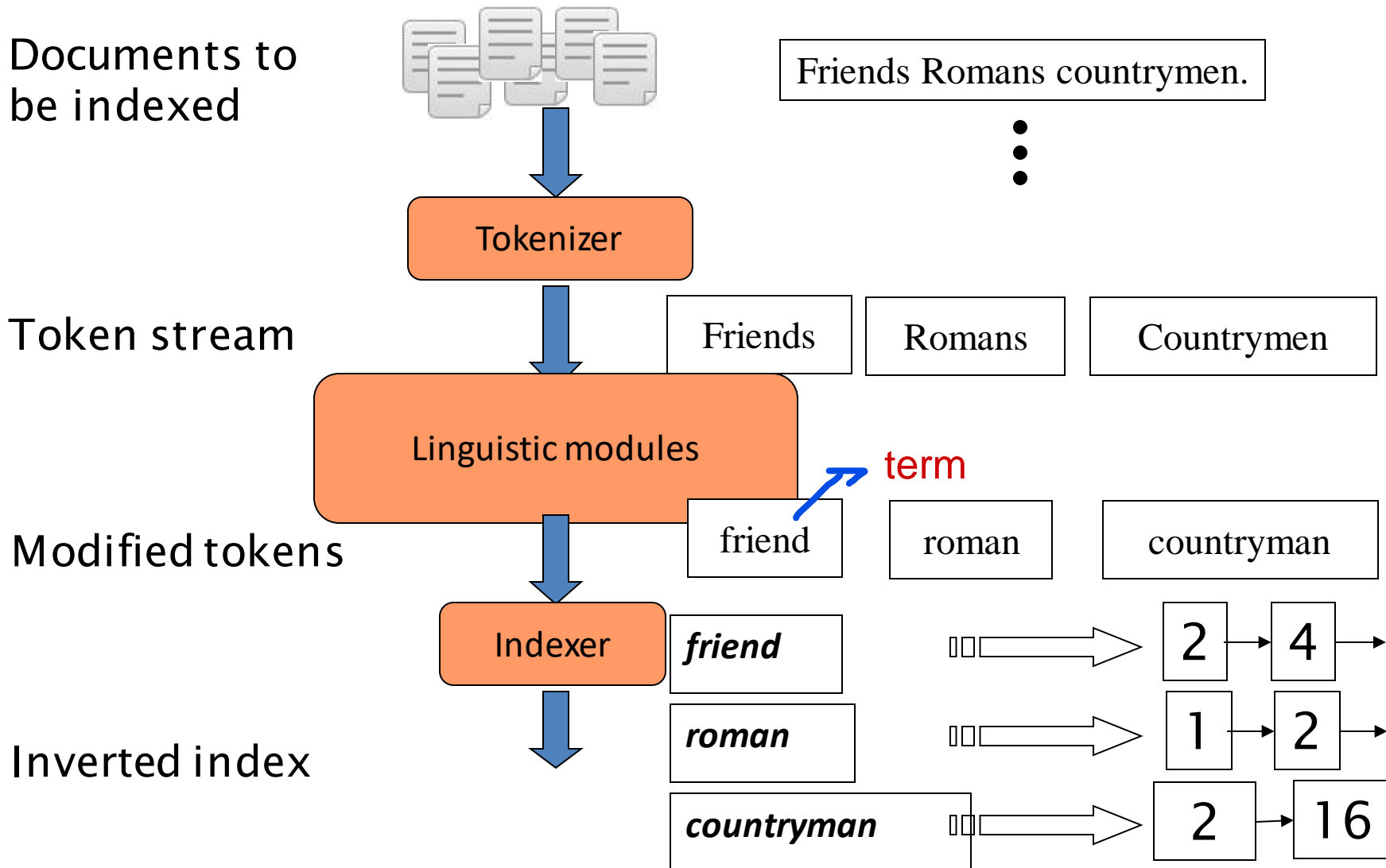
Inverted index

- 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



Initial stages of text processing

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

Indexer steps: Token sequence

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

Doc 1

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

Doc 2

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



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

Indexer steps: Sort

- Sort by terms
 - And then docID



Core indexing step

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

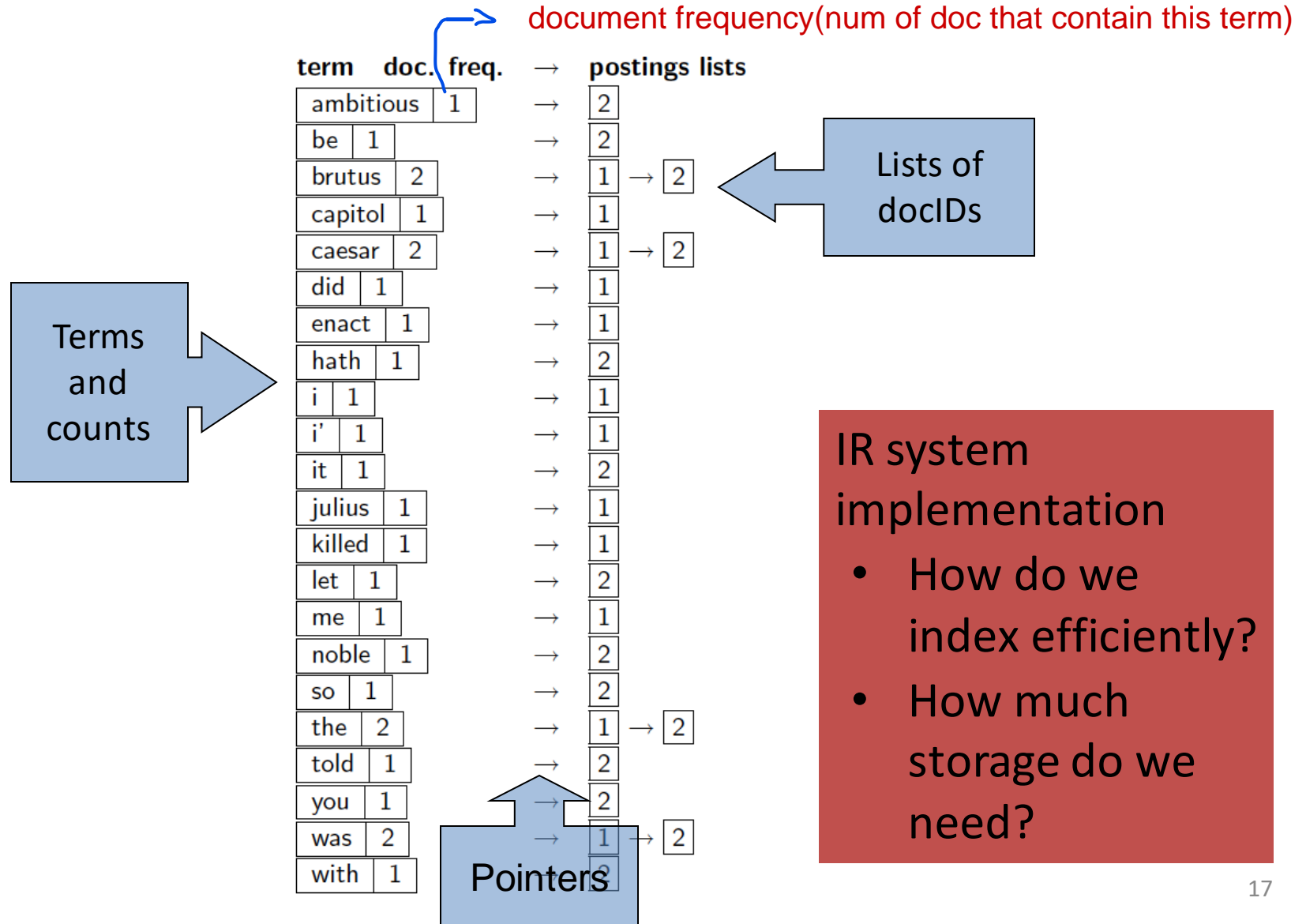
Why frequency?
Will discuss later.

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?



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

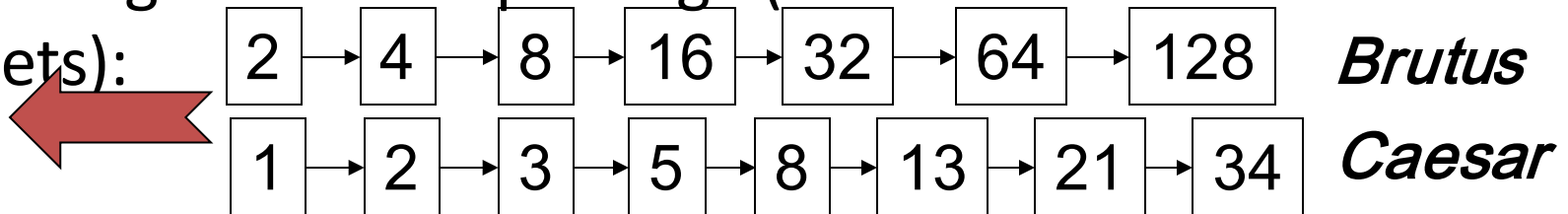
- Locate ***Brutus*** in the Dictionary;

- Retrieve its postings.

- Locate ***Caesar*** in the Dictionary;

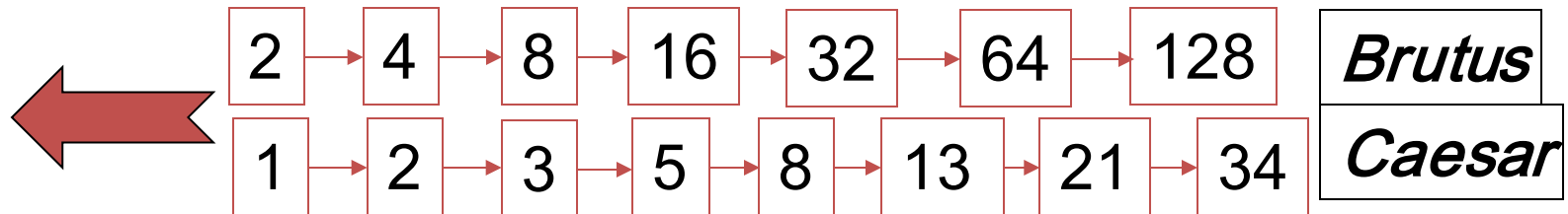
- Retrieve its postings.

- “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
- Primary commercial retrieval tool for 3 decades.
- Many search systems you still use are Boolean:
 - Email, library catalog, Mac OS X Spotlight

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

- Largest commercial (paying subscribers) legal search service (started 1975; ranking added 1992; new federated search added 2010)
- 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

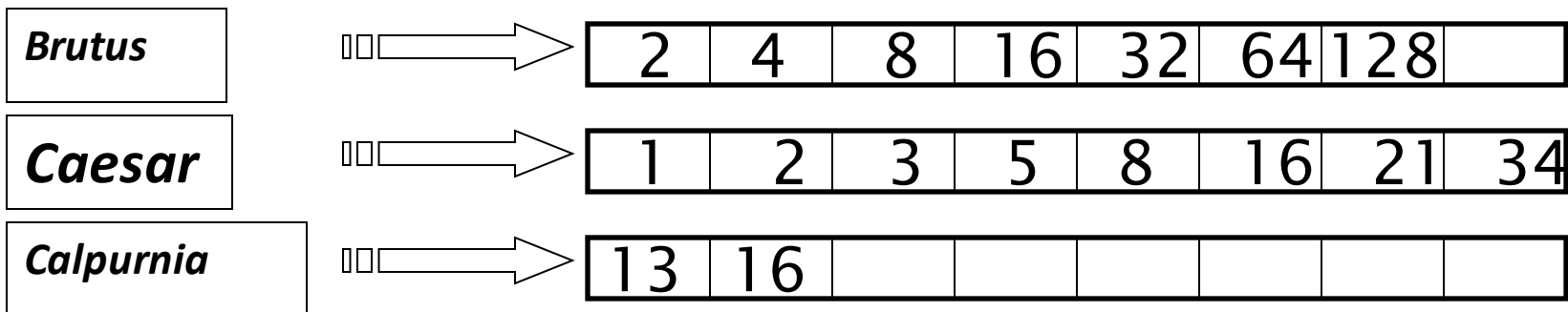
Example: WestLaw

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

- Another example query:
 - Requirements for disabled people to be able to access a workplace
 - disabl! /p access! /s work-site work-place
(employment /3 place
- Note that SPACE is disjunction, not conjunction!
- Long, precise queries; proximity operators; incrementally developed; not like web search
- Many professional searchers still like Boolean search
 - You know exactly what you are getting
- But that doesn't mean it actually works better....

Query optimization

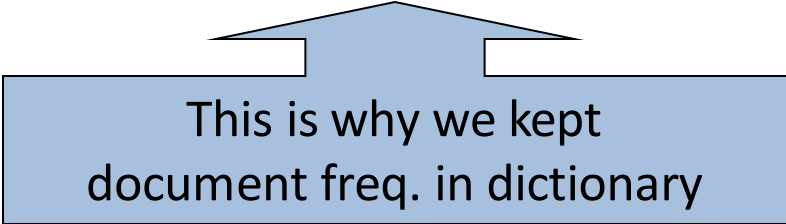
- What is the best order for query processing?
- Consider a query that is an *AND* of n terms.
- For each of the n terms, get its postings, then *AND* them together.



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

Brutus	⇒	2	4	8	16	32	64	128	
Caesar	⇒	1	2	3	5	8	16	21	34
Calpurnia	⇒	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.

Optimized intersection algorithm for conjunctive queries

```
INTERSECT( $\langle t_1, \dots, t_n \rangle$ )  
1   $terms \leftarrow \text{SORTBYINCREASINGFREQUENCY}(\langle t_1, \dots, t_n \rangle)$   
2   $result \leftarrow \text{postings}(\text{first}(terms))$   
3   $terms \leftarrow \text{rest}(terms)$   
4  while  $terms \neq \text{NIL}$  and  $result \neq \text{NIL}$   
5  do  $result \leftarrow \text{INTERSECT}(result, \text{postings}(\text{first}(terms)))$   
6      $terms \leftarrow \text{rest}(terms)$   
7  return  $result$ 
```

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