Introduction to
Information Retrieval

Boolean Retrieval

Terminology

- In the context of a user interacting with an IR system
  - Document: unit of retrieval
  - Each document has a Doc Id
  - Corpus: collection of documents
  - User has information need
  - User inputs a query to system
  - Term: a unit of information (e.g., a word/phrase)
  - Relevance of documents to query/info need
- Ad hoc retrieval task

For most of this lecture

- Corpus: collection of plays of Shakespeare

- Document: an individual play

- Query: a Boolean expression having terms connected with Boolean operators (AND, OR, NOT)

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: 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
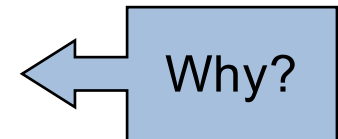
Capitol; **Brutus** killed me.

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.

Why?

- 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?

Brutus	⇒	1	2	4	11	31	45	173	174

Caesar	⇒	1	2	4	5	6	16	57	132

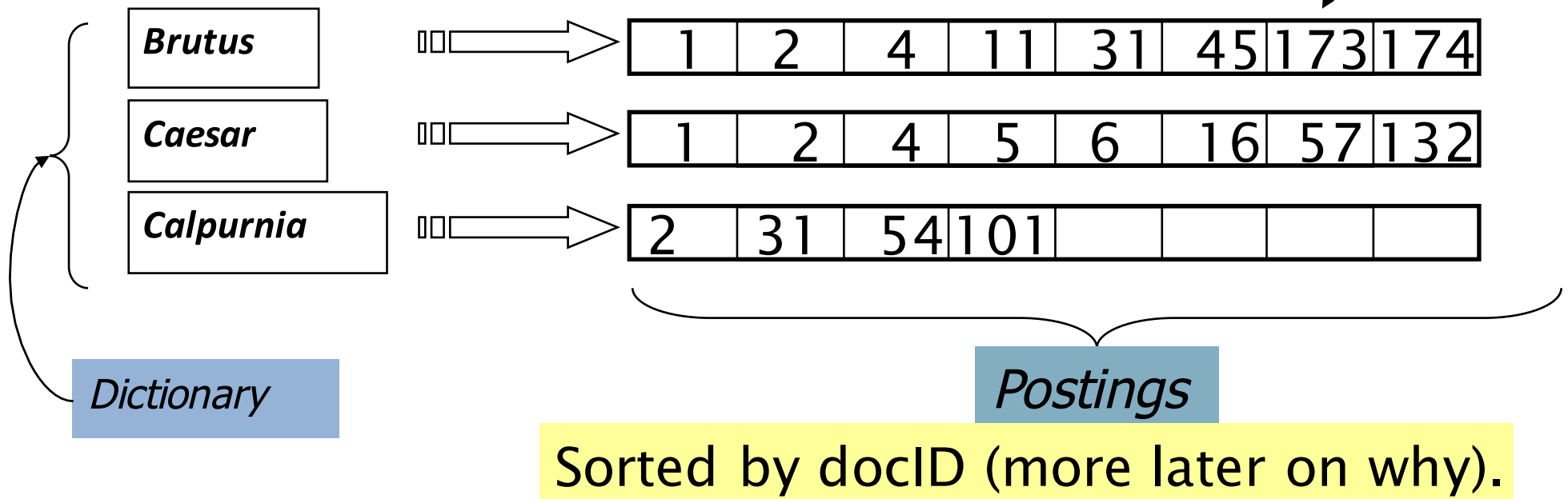| *Calpurnia*	⇒	2	31	54	101				
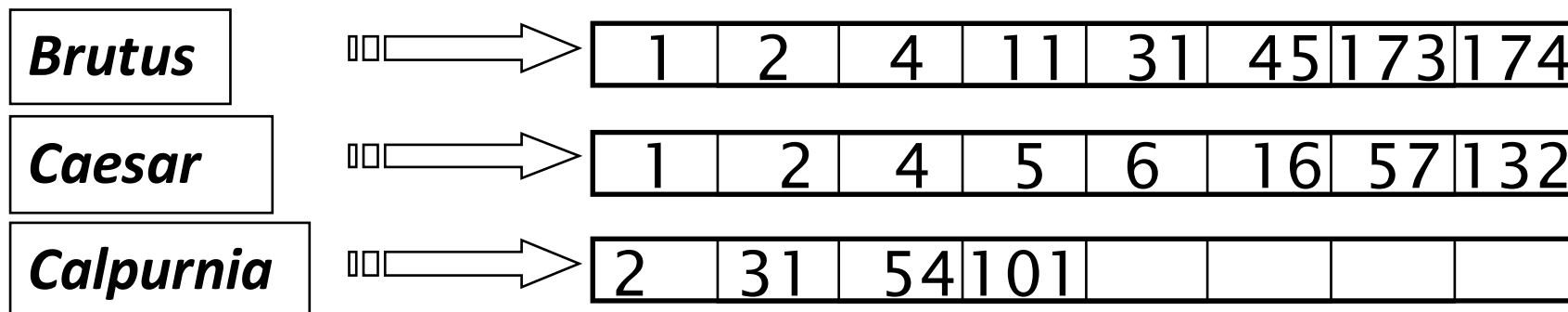
What happens if the word ***Caesar*** is added to document 14?

11

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

Posting

Brutus	→	1	2	4	11	31	45	173	174
Caesar	→	1	2	4	5	6	16	57	132
Calpurnia	→	2	31	54	101				

Dictionary

Postings

Sorted by docID (more later on why).

Inverted index construction

Documents to be indexed

Friends, Romans, countrymen.

Tokenizer

Token stream

Friends	Romans	Countrymen

Linguistic modules

Modified tokens

friend	roman	countryman

Indexer

Inverted index

friend	→	2	4
roman	→	1	2
countryman	→	13	16
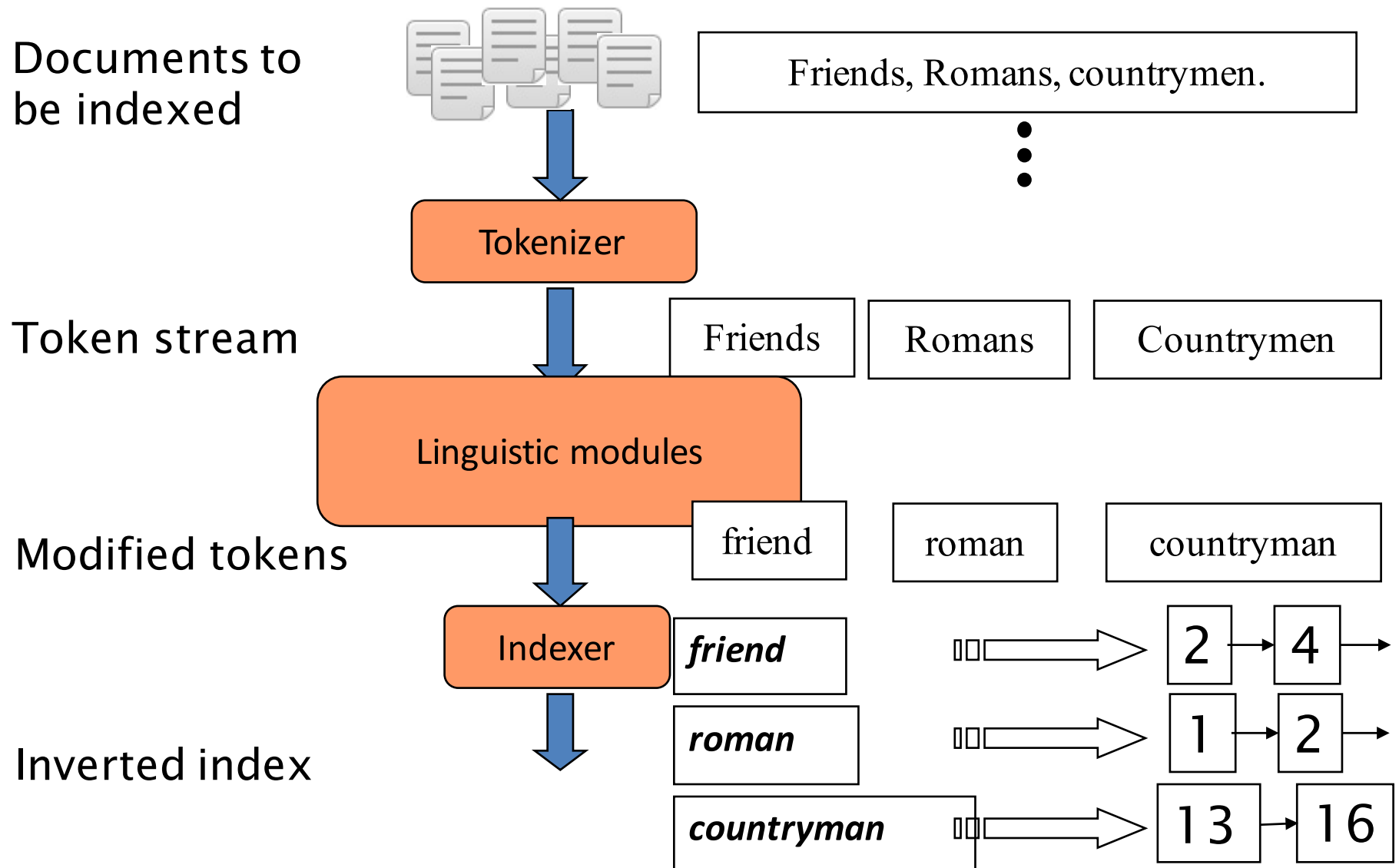
Initial stages of text processing

- 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
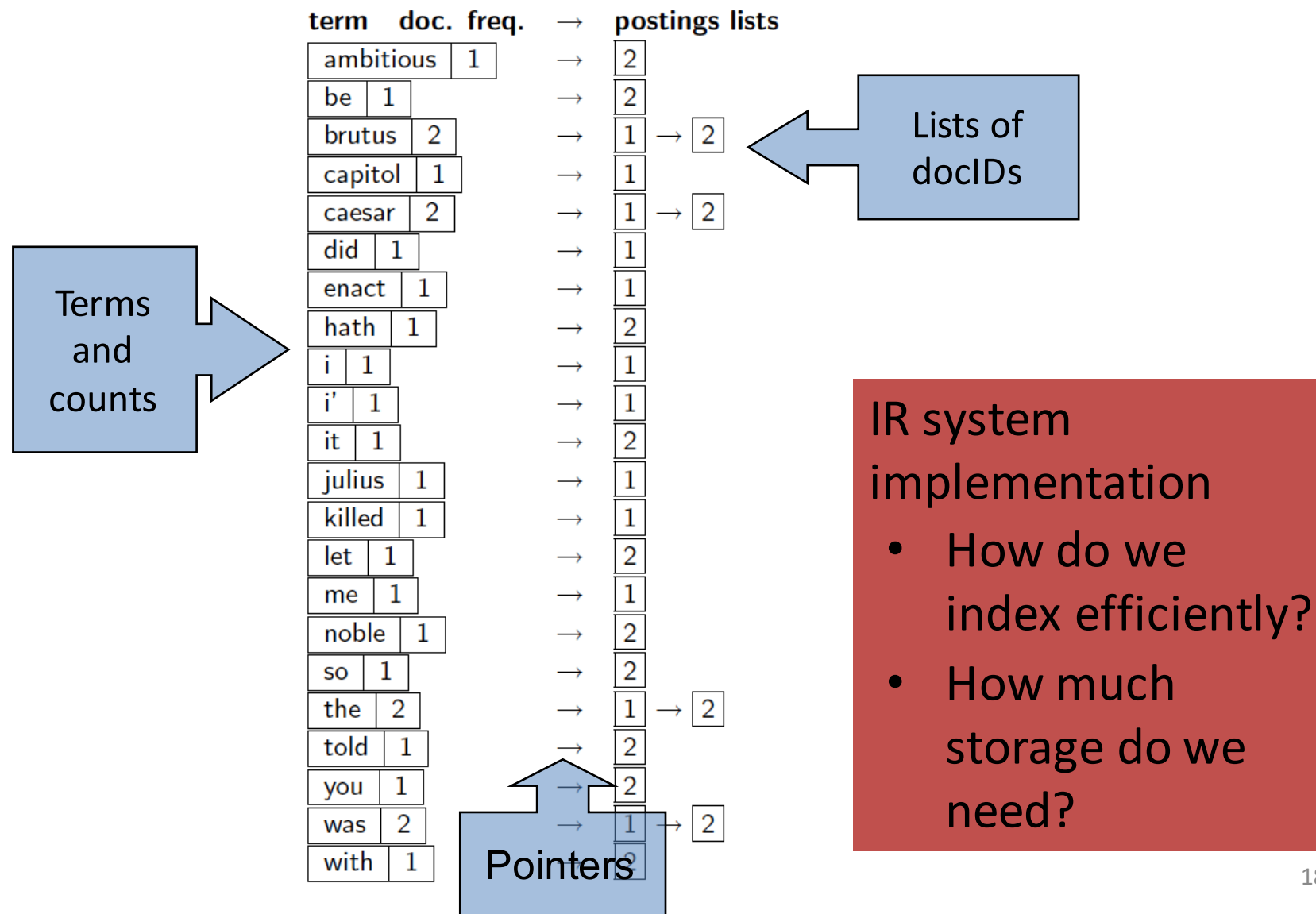- Document frequency information is added to dictionary.

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?

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

Lists of docIDs

Terms and counts

Pointers

IR system implementation
- How do we index efficiently?
- How much storage do we need?

18

Practical considerations

- For a practical IR system handling a huge corpus

- The dictionary will be stored in the memory

- Postings lists will be stored on disk

- Ideally, retrieve (from disk) only those postings lists that are needed to answer a query

Introduction to
Information Retrieval

Query processing with an inverted index

The index we just built

- How do we process a query? $\longleftarrow$ Our focus
  - Later - what kinds of queries can we process?
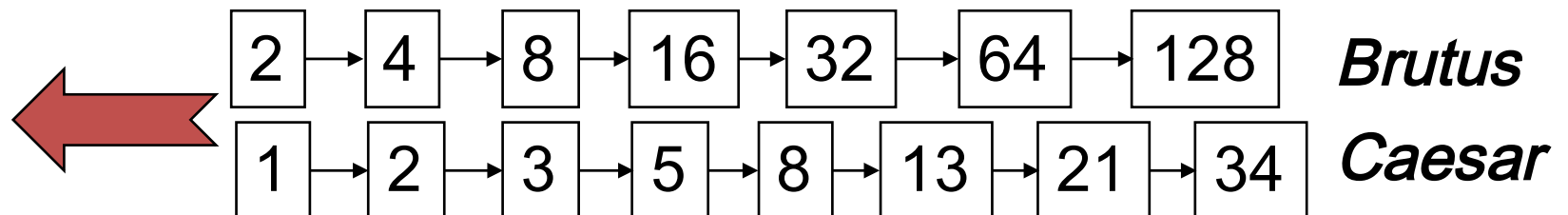
Brutus AND *Caesar*

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):

2	4	8	16	32	64	128	*Brutus*
1	2	3	5	8	13	21	34	*Caesar*

The merge

- Walk through the two postings simultaneously, in time linear in the total number of postings entries

2	4	8	16	32	64	128	*Brutus*
1	2	3	5	8	13	21	34	*Caesar*
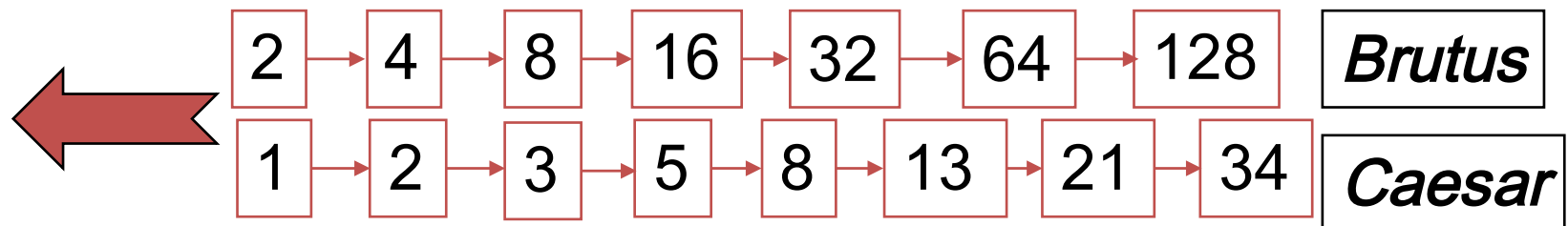
If the list lengths are *x* and *y*, the merge takes O(*x+y*) operations.
Crucial: postings sorted by docID.

Intersecting two postings lists
(a "merge" algorithm)

$\textsc{Intersect}(p_1, p_2)$

1 $answer \leftarrow \langle \, \rangle$
2 **while** $p_1 \neq \textsc{nil}$ and $p_2 \neq \textsc{nil}$
3 **do if** $docID(p_1) = docID(p_2)$
4   **then** $\textsc{Add}(answer, docID(p_1))$
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** $answer$

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 <u>set</u> 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

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

Boolean queries:
More general merges

- Exercise: Adapt the merge for the query:

Brutus *AND NOT* **Caesar**

- Can we still run through the merge in time O($x+y$)?   What can we achieve?

Query optimization

- What is the best order for query processing?

- Consider a query that is an *AND* of *n* terms.

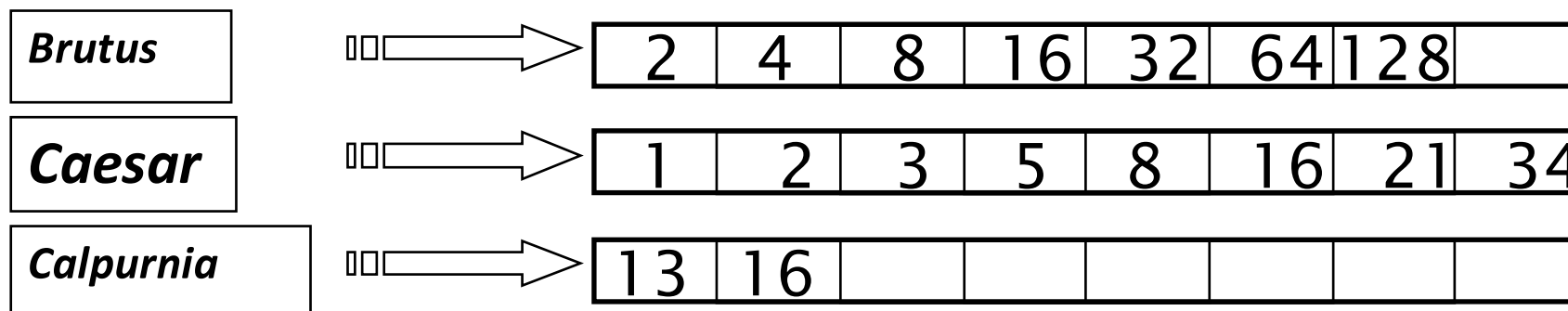- For each of the *n* terms, get its postings, then *AND* them together.

Brutus		2	4	8	16	32	64	128	

Caesar		1	2	3	5	8	16	21	34
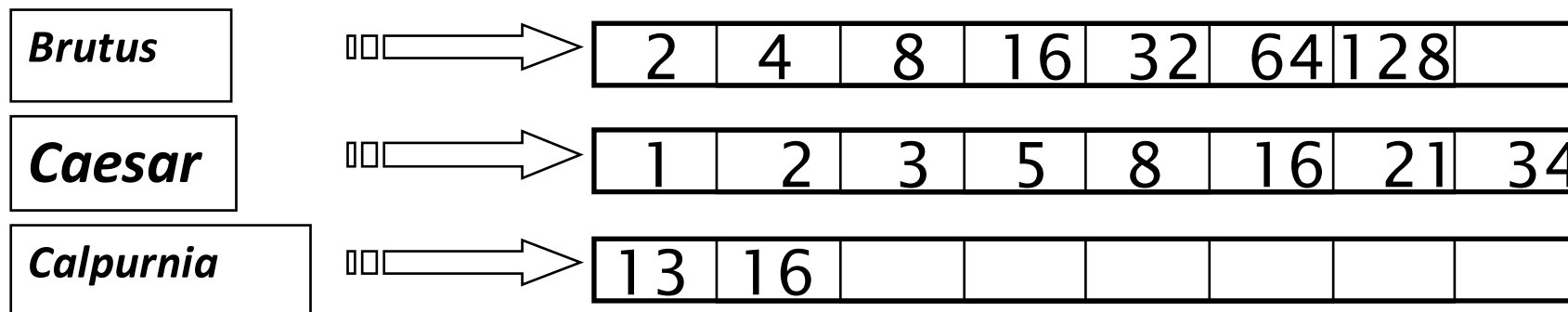
Calpurnia		13	16						

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

28

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.

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

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

Does Google use the Boolean model?

- On Google, the default interpretation of a query $[w_1\ w_2\ .\ .\ .w_n]$ is w_1 AND w_2 AND . . .AND w_n
- Cases where you get hits that do not contain one of the wi :
  - anchor text
  - page contains variant of w_i (morphology, spelling correction, synonym)
  - long queries (n large)
  - boolean expression generates very few hits
- Simple Boolean vs. Ranking of result set
  - Simple Boolean retrieval returns matching documents in no particular order.
  - Google (and most well designed Boolean engines) rank the result set – they rank good hits (according to some estimator of relevance) higher than bad hits.