

Introduction to **Information Retrieval**

Lecture 1: Boolean retrieval

Unstructured data in 1680

- 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

	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.
- $110100 \text{ AND } 110111 \text{ AND } 101111 = 100100$.

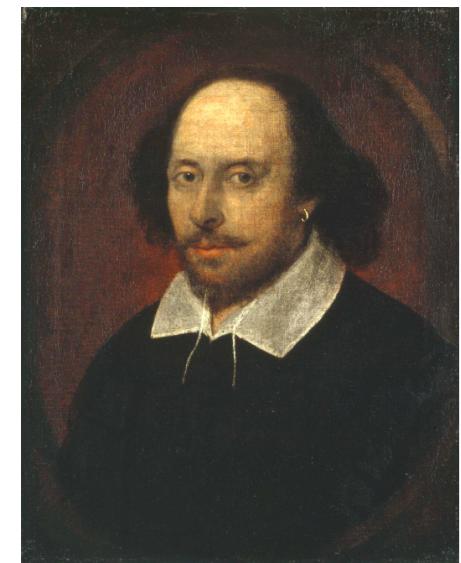
Answers to query

- 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

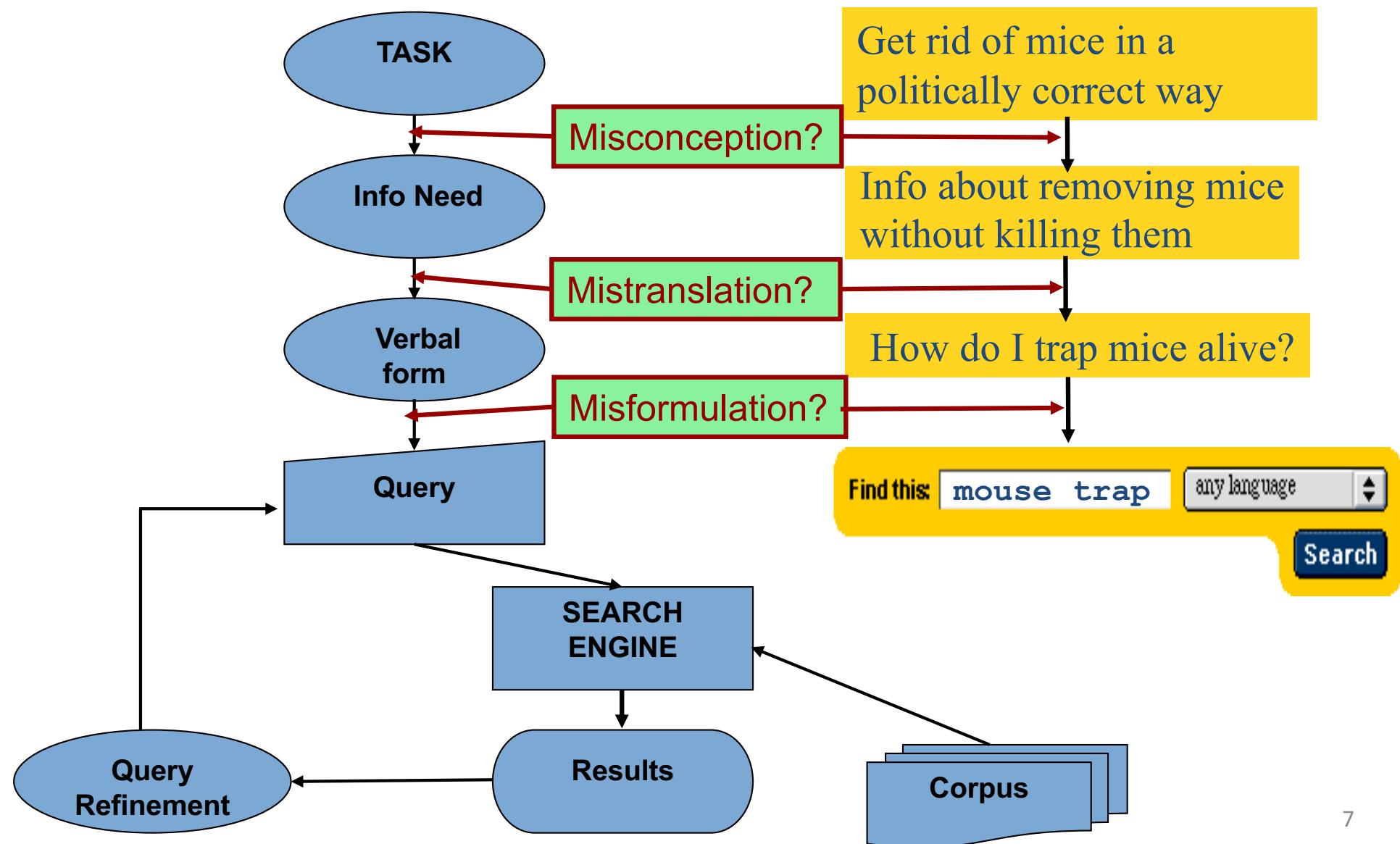
Lord Polonius: I did enact Julius **Caesar** I was killed i' the
Capitol; **Brutus** killed me.



Basic assumptions of Information Retrieval

- **Collection:** Fixed set of documents
- **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?

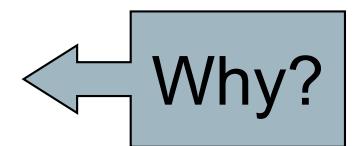
- *Precision* : Fraction of retrieved docs that are relevant to user's information need
- *Recall* : Fraction of relevant docs in collection that are retrieved
- More precise definitions and measurements to follow in later lectures

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.

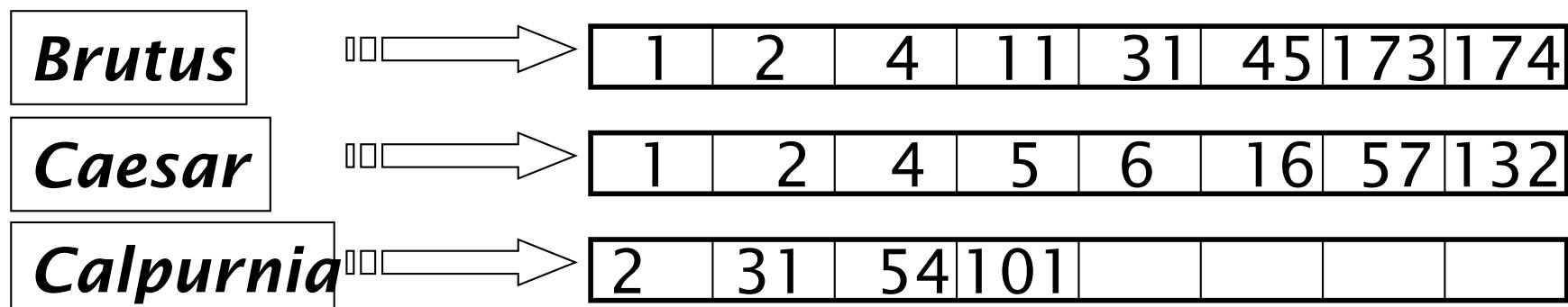
Can't build the matrix

- 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

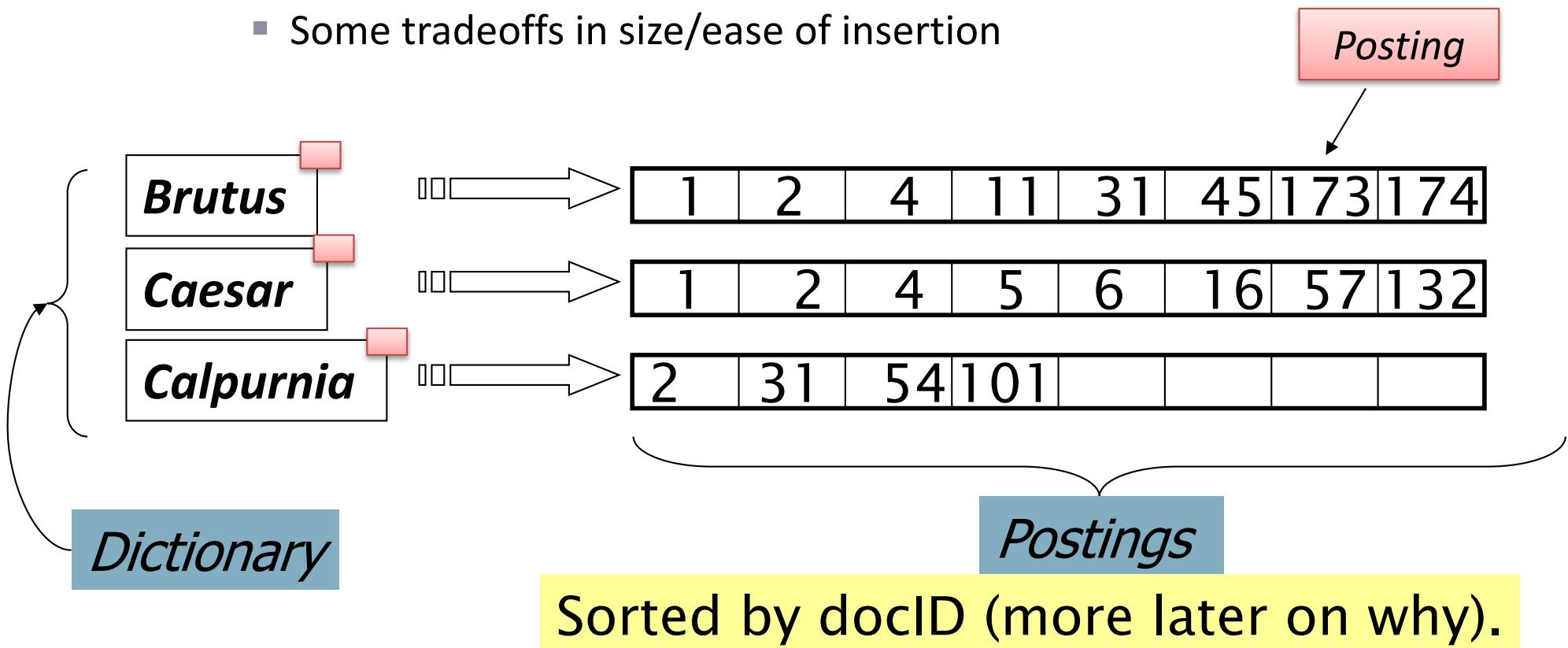
- For each term t , we must store a list of all documents that contain t .
 - Identify each by a **docID**, a document serial number
- Can we used fixed-size arrays for this?



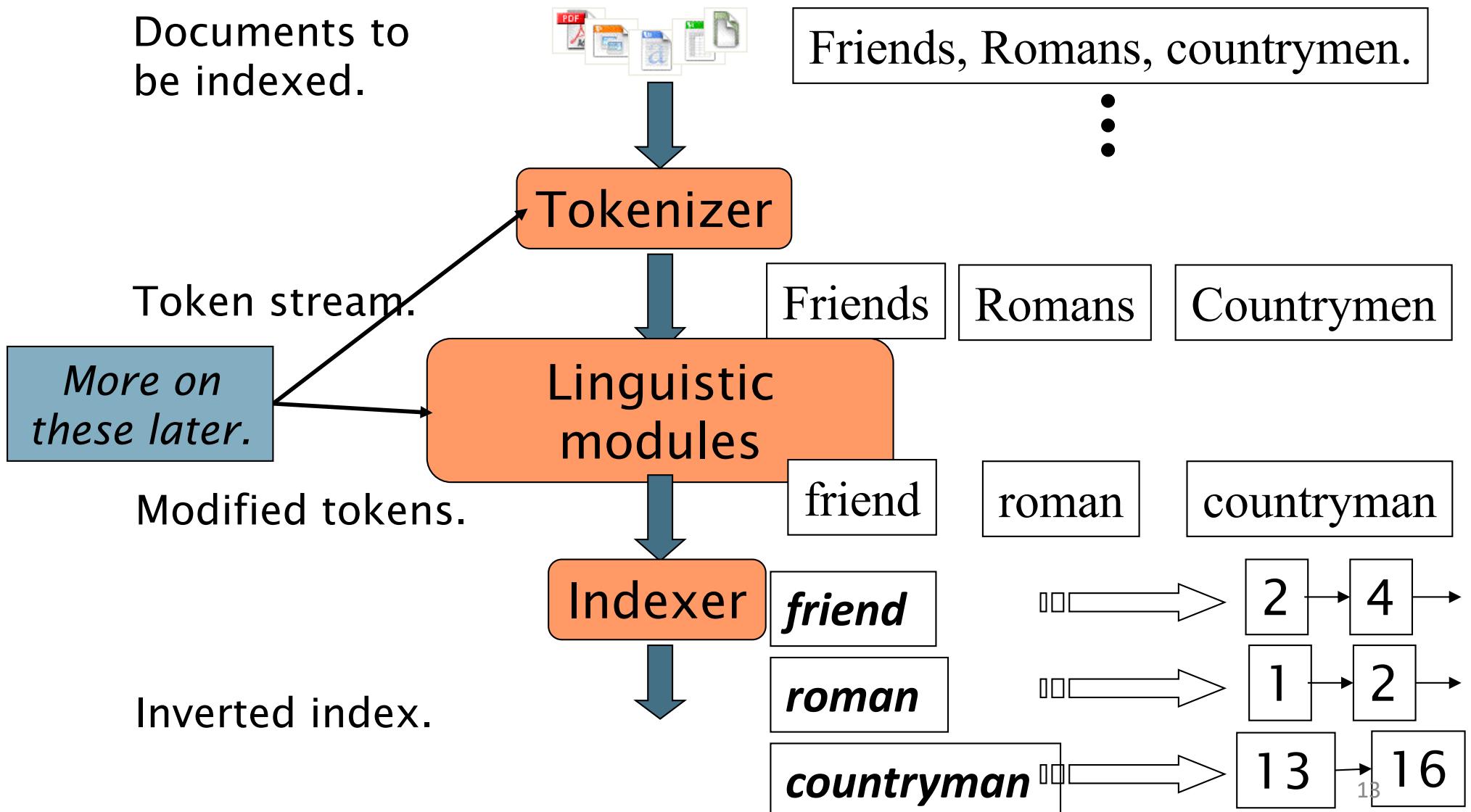
What happens if the word ***Caesar*** is added to document 14?

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



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



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.

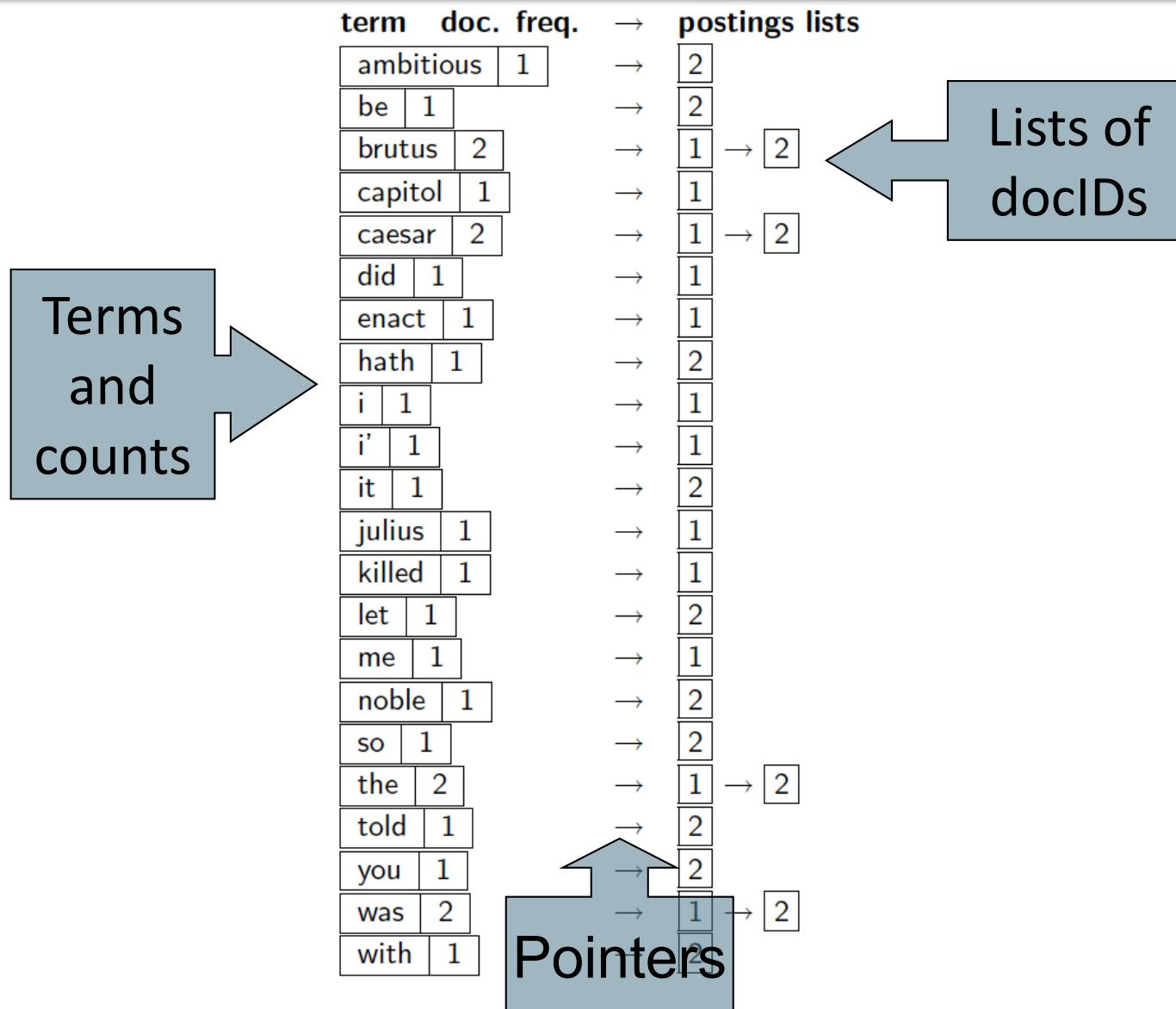
Why frequency?
Will discuss later.

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

Where do we pay in storage?

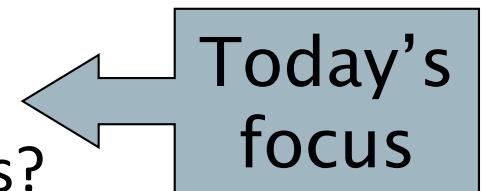


More on Inverted Index Construction

- Hashing-based construction methods are more efficient (though harder to implement)
- Inverted index can be compressed to
 - reduce index size
 - reduces transfer time between storage and memory

The index we just built

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

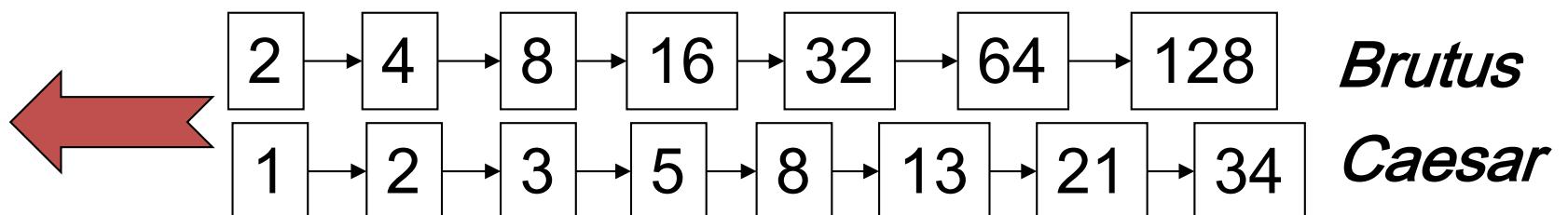


Query processing: AND

- Consider processing the query:

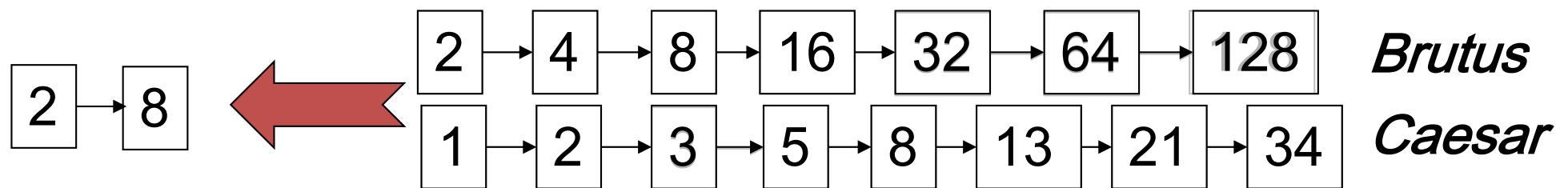
Brutus AND Caesar

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



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** 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 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

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

- Largest commercial (paying subscribers) legal search service (started 1975; ranking added 1992)
- 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**
 - **foo! = foo*, /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....

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?

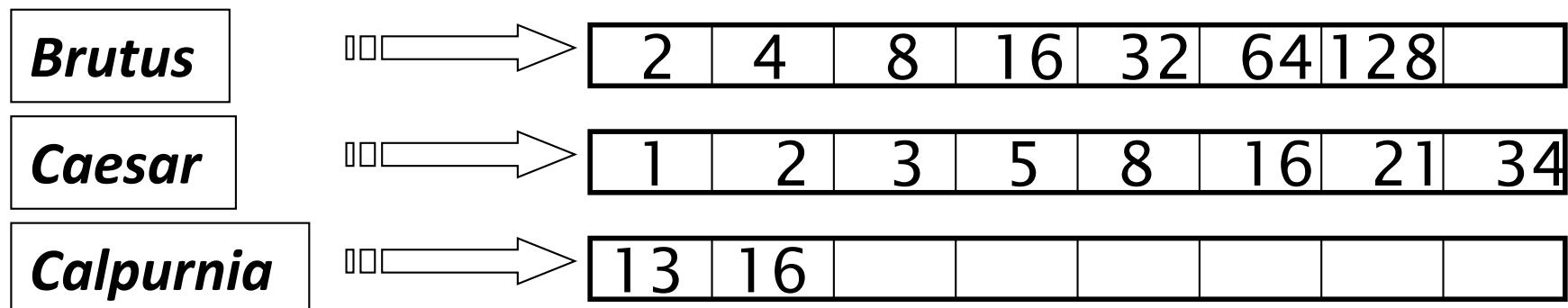
(Brutus OR Caesar) AND NOT

(Antony OR Cleopatra)

- 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* them together.



Query: **Brutus AND Calpurnia AND Caesar**

Query optimization example

- Process in order of increasing freq:
 - *start with the 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) AND (light OR lord)**
- 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)*

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

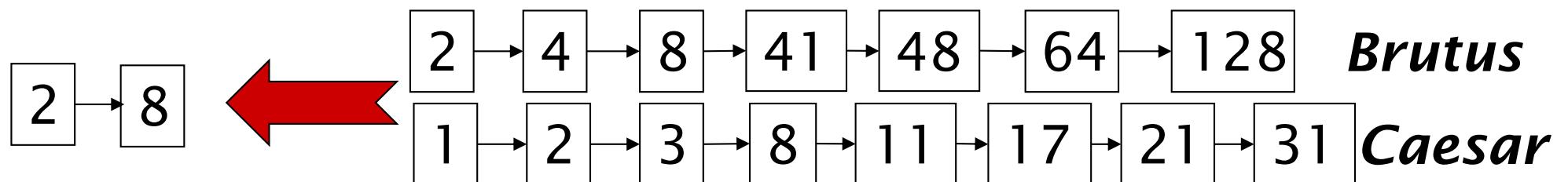
Q: Any more accurate way to estimate the cardinality of intermediate results?

Q: Can we merge multiple lists (>2) simultaneously?

FASTER POSTINGS MERGES: SKIP POINTERS/SKIP LISTS

Recall basic merge

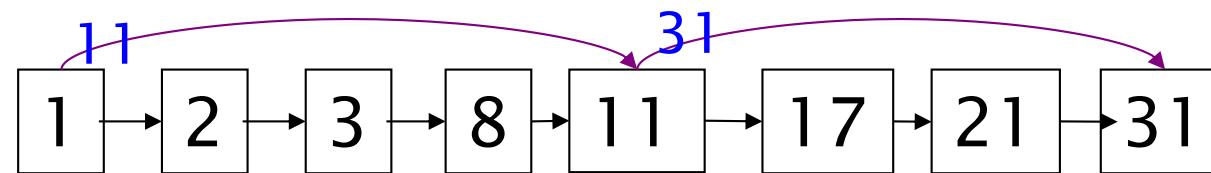
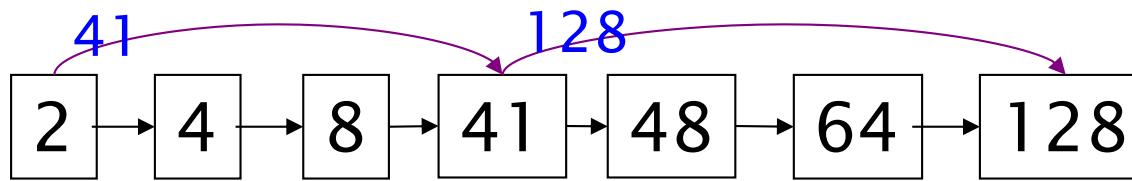
- Walk through the two postings simultaneously, in time linear in the total number of postings entries



If the list lengths are m and n , the merge takes $O(m+n)$ operations.

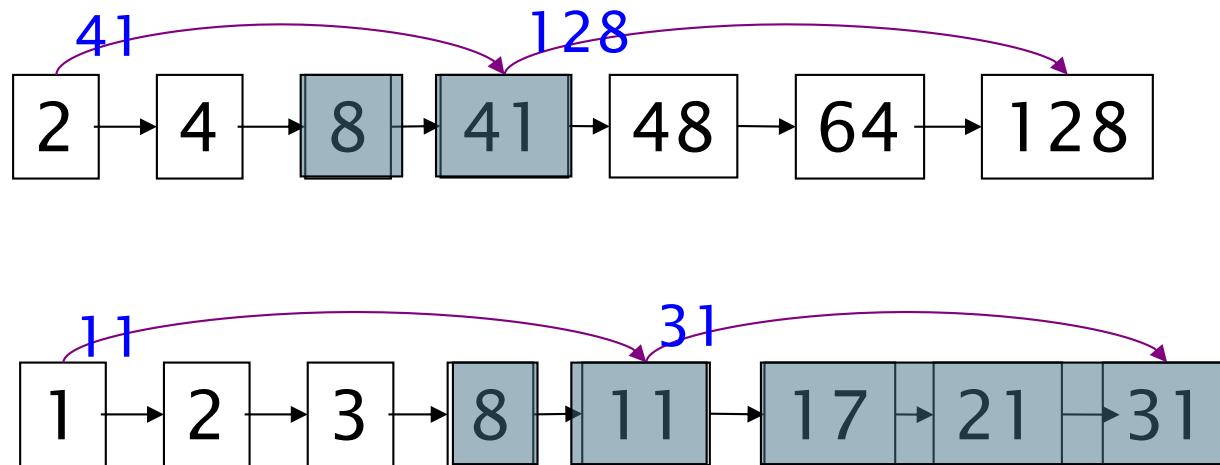
Can we do better?
Yes (if index isn't changing too fast).

Augment postings with skip pointers (at indexing time)



- Why?
- To skip postings that will not figure in the search results.
- How?
- Where do we place skip pointers?

Query processing with skip pointers



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

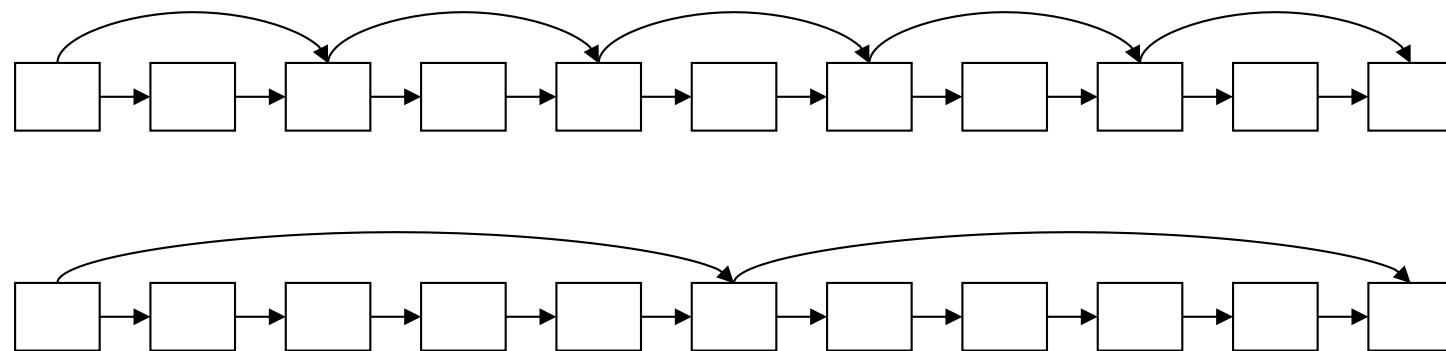
We then have 41 and 11 on the lower. 11 is smaller.

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

Can we skip w/o skip pointers?

Where do we place skips?

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

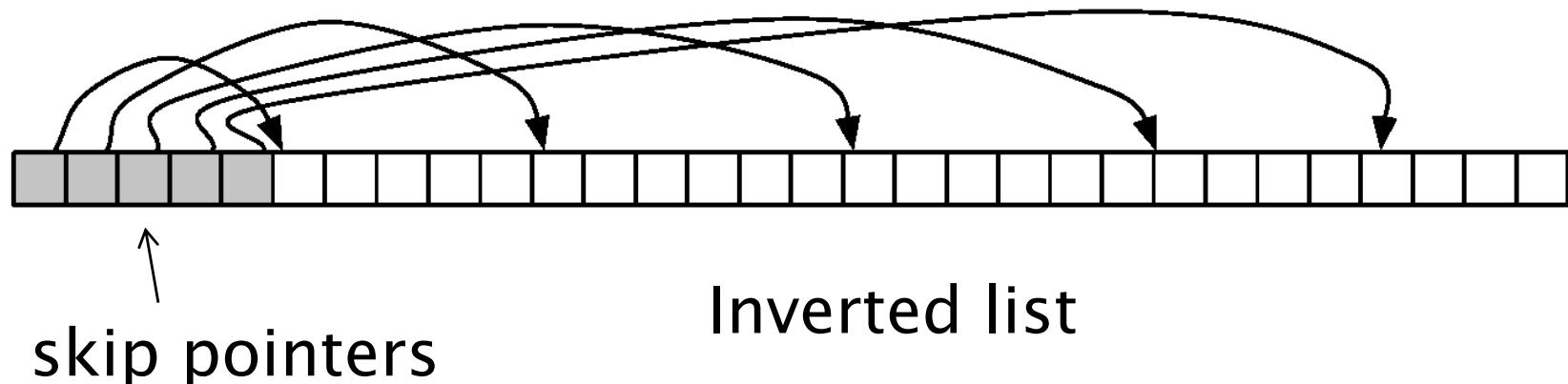


Placing skips

- Simple heuristic: for postings of length L , use $L^{1/2}$ evenly-spaced skip pointers.
 - This ignores the distribution of query terms.
 - Easy if the index is relatively static; harder if L keeps changing because of updates.
-
- This definitely used to help; with modern hardware it may not (Bahle et al. 2002) unless you're memory-based
 - The I/O cost of loading a bigger postings list can outweigh the gains from quicker in memory merging!

Skip Pointers

- A skip pointer (d, p) contains a document number d and a byte (or bit) position p
 - Means there is an inverted list posting that starts at position p , and the posting before it was for document d



PHRASE QUERIES AND POSITIONAL INDEXES

Phrase queries

- 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.
 - The concept of phrase queries has proven easily understood by users; one of the few “advanced search” ideas that works
 - Many more queries are *implicit phrase queries*
- For this, it no longer suffices to store only $\langle \text{term} : \text{docs} \rangle$ entries

Solution 1: 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 are processed as we did with wild-cards:
- ***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!

Extended biwords

- Parse the indexed text and perform part-of-speech-tagging (POST).
- Bucket the terms into (say) Nouns (N) and articles/prepositions (X).
- Call any string of terms of the form NX*N an extended biword.
 - Each such extended biword is now made a term in the dictionary.
- Example: ***catcher in the rye***

N X X N

- Query processing: parse it into N's and X's
 - Segment query into enhanced biwords
 - Look up in index: ***catcher rye***

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***;

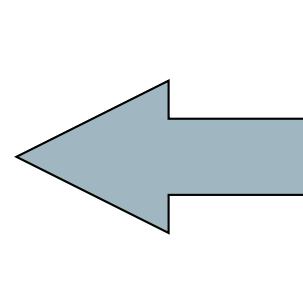
doc1: position1, position2 ... ;

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; ...
- Same general method for proximity searches

Proximity queries

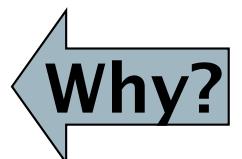
- **LIMIT! /3 STATUTE /3 FEDERAL /2 TORT**
 - Again, here, $/k$ means “within k words of”.
- Clearly, positional indexes can be used for such queries; biword indexes cannot.
- Exercise: Adapt the linear merge of postings to handle proximity queries. Can you make it work for any value of k ?
 - This is a little tricky to do correctly and efficiently
 - See Figure 2.12 of IIR (Page 39)
 - There’s likely to be a problem on it!

Positional index size

- You can compress position values/offsets: we'll talk about that in lecture 5
- Nevertheless, a positional index expands postings storage *substantially*
- 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.

Positional index size

- Need an entry for **each occurrence**, not just once per document
- Index size depends on average document size
 - Average web page has <1000 terms
 - SEC filings, books, even some epic poems ... easily 100,000 terms
- Consider a term with frequency 0.1%



Document size	Postings	Positional postings
1000	1	1
100,000	1	100

Rules of thumb

- A positional index is 2–4 as large as a non-positional index
- Positional index size 35–50% of volume of original text
- Caveat: all of this holds for “English-like” languages

Combination schemes

- These two approaches can be profitably combined
 - For particular phrases (***“Michael Jackson”, “Britney Spears”***) it is inefficient to keep on merging positional postings lists
 - Even more so for phrases like ***“The Who”***
- Williams et al. (2004) evaluate a more sophisticated mixed indexing scheme
 - A typical web query mixture was executed in $\frac{1}{4}$ of the time of using just a positional index
 - It required 26% more space than having a positional index alone

[Optional]

\$ < \text{any char}

Solution 3: Suffix Tree/Array

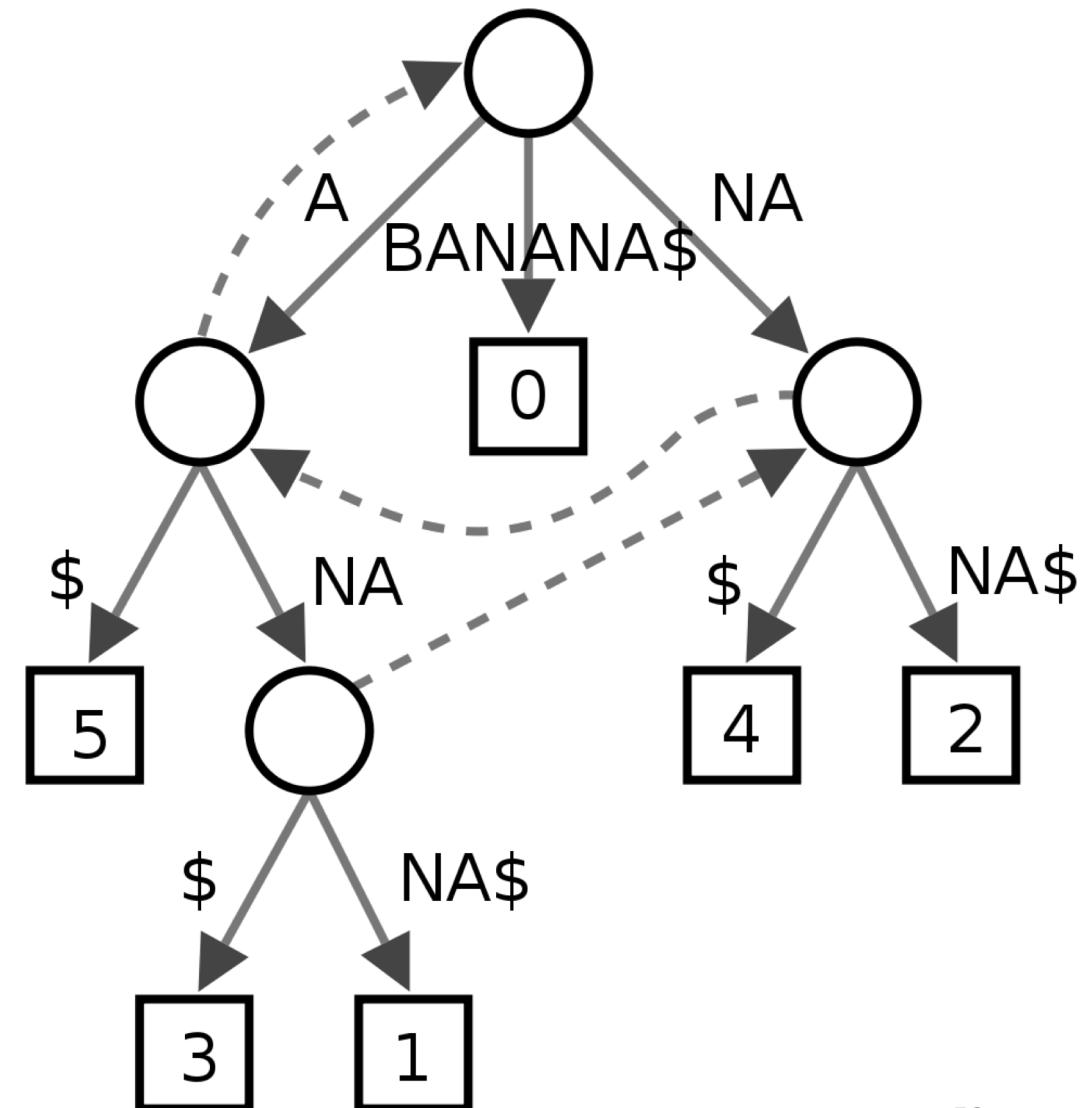
- BANANA\$

 - BANANA\$ pos:0
 - ANANA\$ pos:1
 - NANA\$ pos:2
 - ANA\$ pos:3
 - NA\$ pos:4
 - A\$ pos:5



Sort on the strings

- A\$ pos:5
- ANA\$ pos:3
- ANANA\$ pos:1
- BANANA\$ pos:0
- NA\$ pos:4
- NANA\$ pos:2



[Optional]

\$ < any char

Suffix Array

- BANANA\$
 - BANANA\$ pos:0
 - ANANA\$ pos:1
 - NANA\$ pos:2
 - ANA\$ pos:3
 - NA\$ pos:4
 - A\$ pos:5
 - \$ pos:6

↓

 - \$ pos:6
 - A\$ pos:5
 - ANA\$ pos:3
 - ANANA\$ pos:1
 - BANANA\$ pos:0
 - NA\$ pos:4
 - NANA\$ pos:2
- Sort on the strings
-
- | | | | | | | |
|---|---|---|---|---|---|----|
| B | A | N | A | N | A | \$ |
| 6 | 5 | 3 | 1 | 0 | 4 | 2 |
- ← Suffix array
- Binary search (using offsets to fetch the ‘key’)

Resources for today's lecture

- *Introduction to Information Retrieval*, chapter 1
- Shakespeare:
 - <http://www.rhymezone.com/shakespeare/>
 - Try the neat browse by keyword sequence feature!
- *Managing Gigabytes*, chapter 3.2
- *Modern Information Retrieval*, chapter 8.2

Resources for today's lecture

- Skip Lists theory: Pugh (1990)
 - Multilevel skip lists give same $O(\log n)$ efficiency as trees
- H.E. Williams, J. Zobel, and D. Bahle. 2004. “Fast Phrase Querying with Combined Indexes”, ACM Transactions on Information Systems.
<http://www.seg.rmit.edu.au/research/research.php?author=4>
- D. Bahle, H. Williams, and J. Zobel. Efficient phrase querying with an auxiliary index. SIGIR 2002, pp. 215-221.