CS 589 Fall 2021 Lecture 4

Inverted Index

Monday 6:30-9:00
Babbio 122

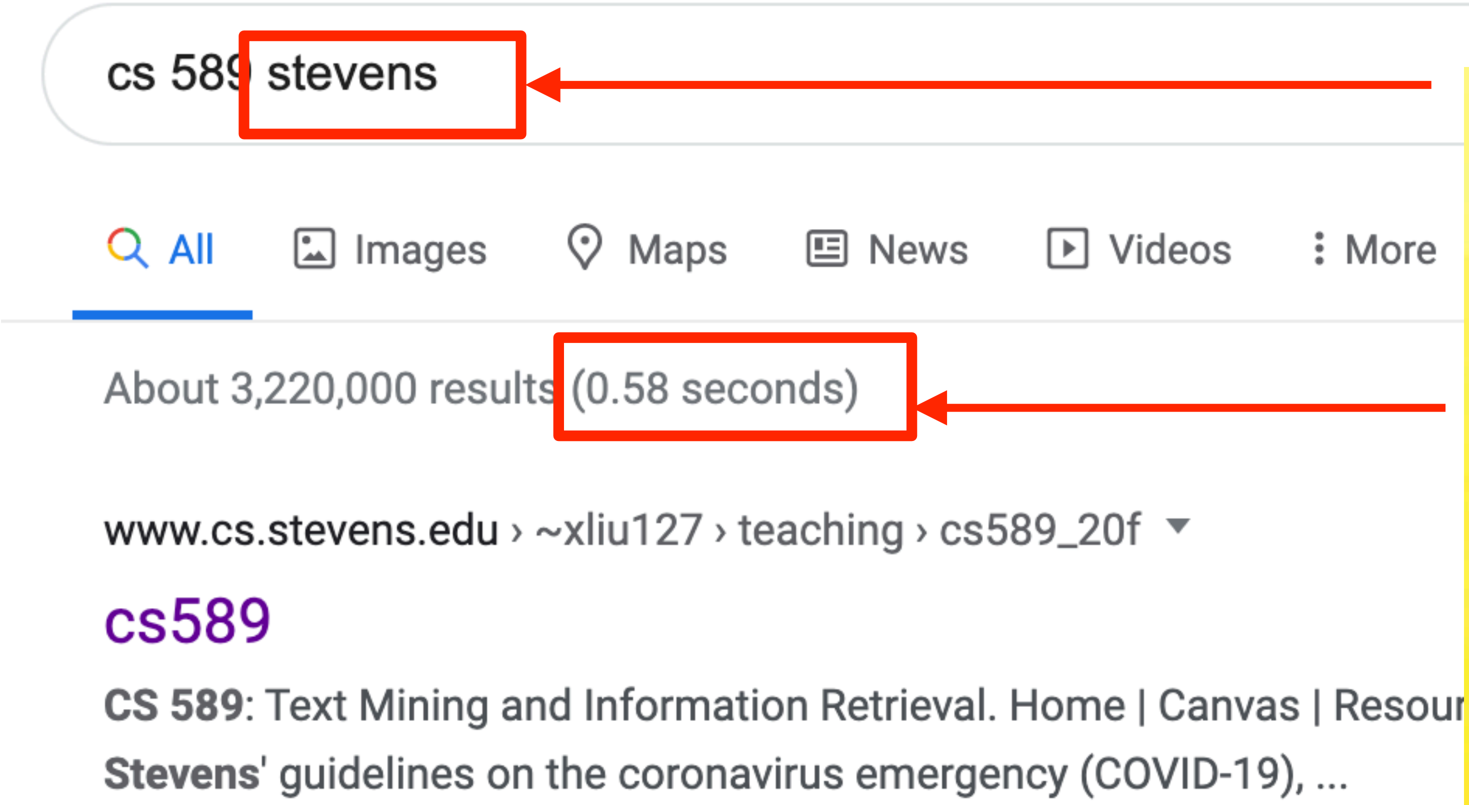
All zoom links in Canvas
Most slides adapted from Stanford CS276

photo: https://www.scubedstudios.com/information-retrieval/

Lecture 4: Information retrieval infrastructure

How does Google know cs 589 refers to a course?

How does Google



cs 589 stevens

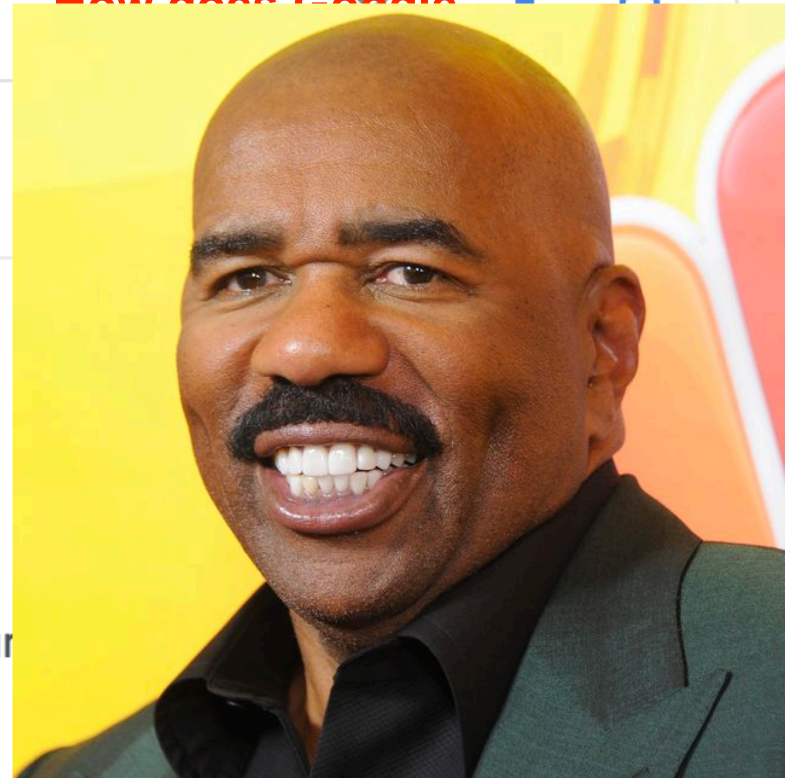
About 3,220,000 results (0.58 seconds)

www.cs.stevens.edu › ~xliu127 › teaching › cs589_20f

cs589

CS 589: Text Mining and Information Retrieval. Home | Canvas | Resour

Stevens' guidelines on the coronavirus emergency (COVID-19), ...



Review of Lecture 1-3: Retrieval models

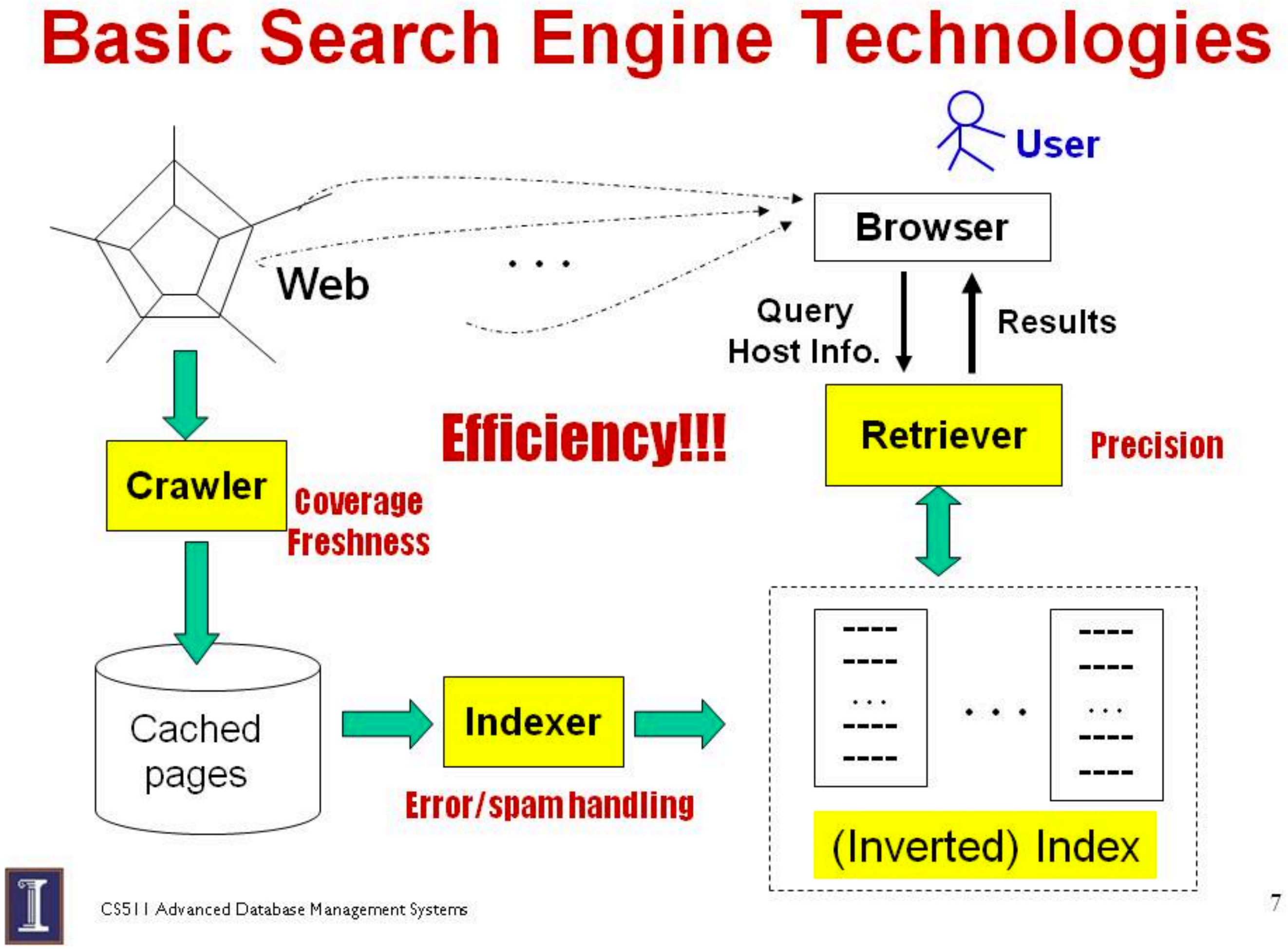
- Vector space model
  - Use the cosine score of TF-IDF to retrieve documents

- BM25:
  - Approximated 2-Poisson model derived from PRP, i.e., rank by $p(rel = 1|q, d)$
  - Adding document length pivoting and IDF

- LM-based retrieval model
  - Rank by p(q|d) based on i.i.d. assumption for words and unigram LM

$$p(w_i|d) = \begin{cases} p_{seen}(w_i|d) & \text{if } w_i \text{ is seen in d} \\ \alpha_d p(w_i|C) & o.w. \end{cases}$$

Lecture 4: Information retrieval infrastructure



Basic Search Engine Technologies

CS511 Advanced Database Management Systems

7

Pop quiz (IR Evaluation)

- Suppose a query has a total of 4 relevant documents in a collection with 100 documents. System A and System B have each retrieved 10 documents, and the relevance status of the two ranked lists of results is:

System A: [+,+,-,-,-,-,-,-,-,-]

System B: [+,-,+,-,-,-,-,-,-,-]

1. What is the MAP of System A and System B?
2. What is the NDCG@5 of System A?

Lecture 4: Inverted index

- Key data structure underlying all modern IR systems
  - Systems run on a single machine
  - Massive systems for the biggest commercial search engines

- Exploiting the sparsity of the term-document matrix

- Inverted index can generally be applied to retrieval models
  - TF-IDF, BM25, LM-based retrieval model

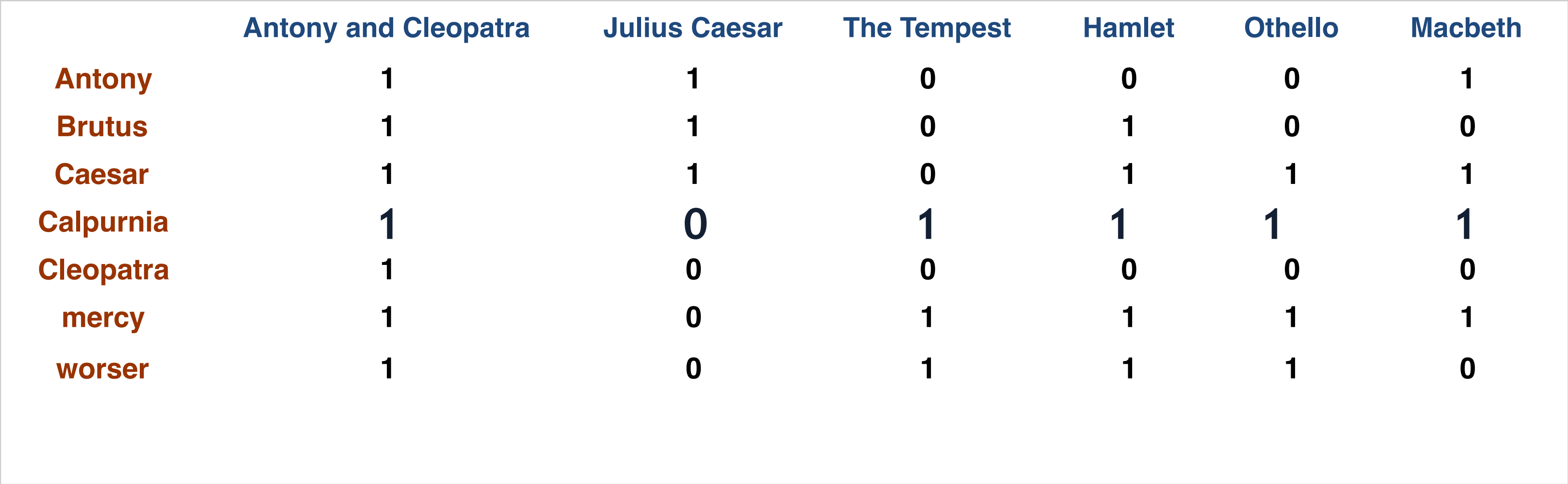
Lecture 4: Motivating example

- Which plays of Shakespeare contain the words ***Brutus*** *AND* ***Caesar*** but *NOT* ***Calpurnia***?

- *grep -r ./* “.* brutus .* caesar.*”* then remove all documents containing *calpurnia*
  - Slow when the data size is large
  - Ranked retrieval

Lecture 4: Motivating example

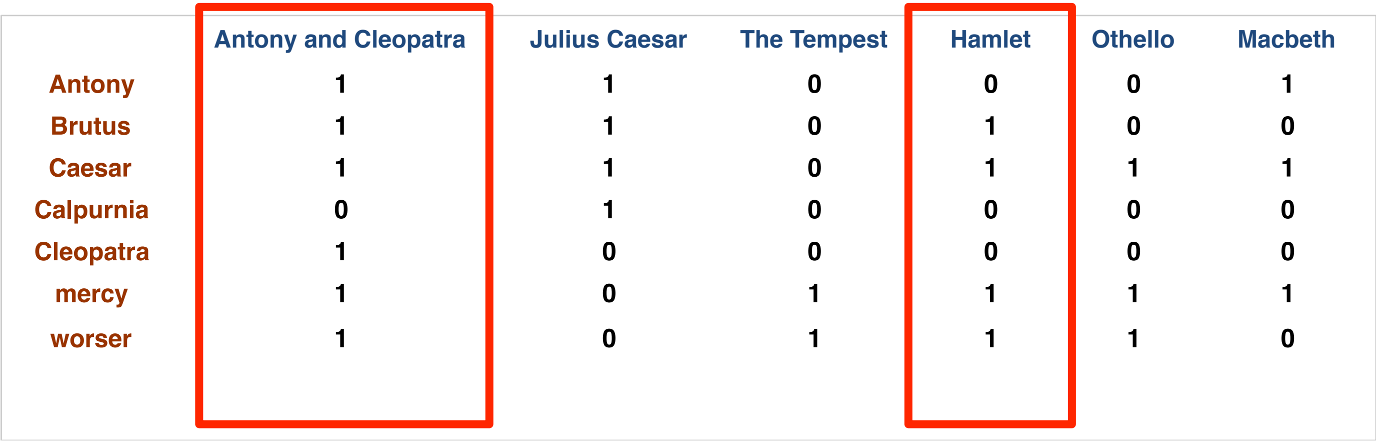
- Which plays of Shakespeare contain the words ***Brutus*** *AND* ***Caesar*** but *NOT* ***Calpurnia***? ***Brutus*** *AND* ***Caesar*** *AND NOT* ***Calpurnia***



	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	1	0	1	1	1	1
Cleopatra	1	0	0	0	0	0
mercy	1	0	1	1	1	1
worser	1	0	1	1	1	0

Lecture 4: Motivating example

- Which plays of Shakespeare contain the words ***Brutus*** *AND* ***Caesar*** but *NOT* ***Calpurnia***? ***Brutus*** *AND* ***Caesar*** *AND NOT* ***Calpurnia***
  - 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

Lecture 4: Motivating example

- Which plays of Shakespeare contain the words ***Brutus*** *AND* ***Caesar*** but *NOT* ***Calpurnia***?

- 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.

What if we have a bigger collection?

- Consider *N* = 1 million documents, each with about 1000 words
  - Avg 6 bytes/word including spaces/punctuation, 6GB

- Say there are *M* = 500K *distinct* terms among these.

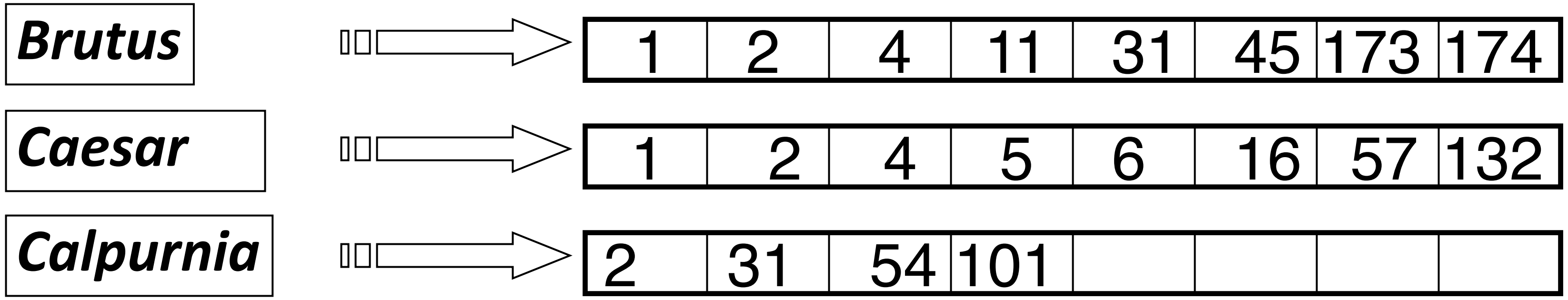
- 500K x 1M matrix has half-a-trillion 0’s and 1’s
  - But it has no more than one billion 1’s (**why?**)

- A better representation: only record the 1’s

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

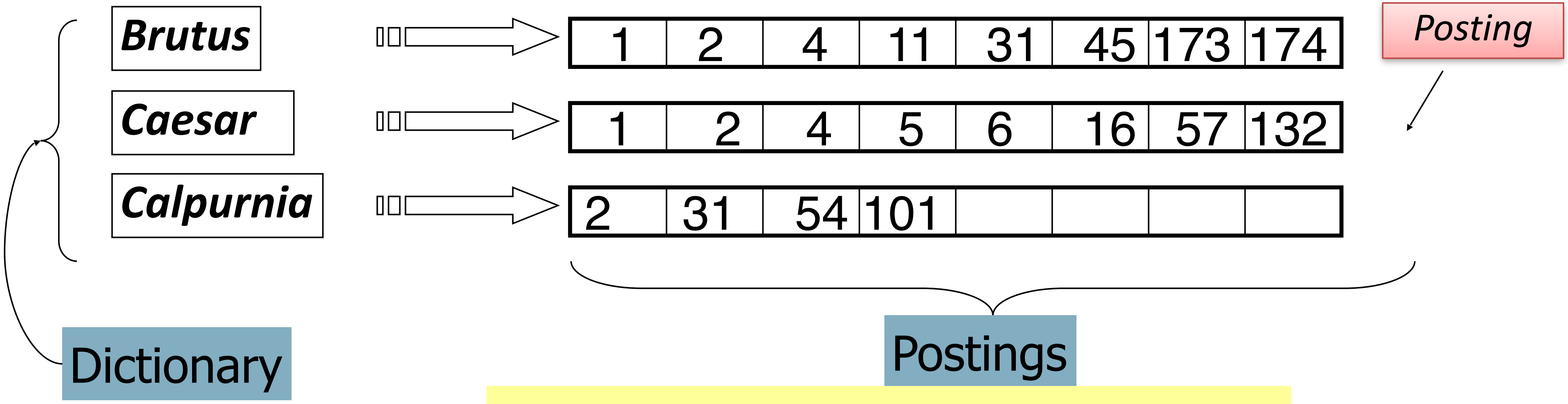
- Can we used **fixed-size arrays** for this?



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

Inverted index

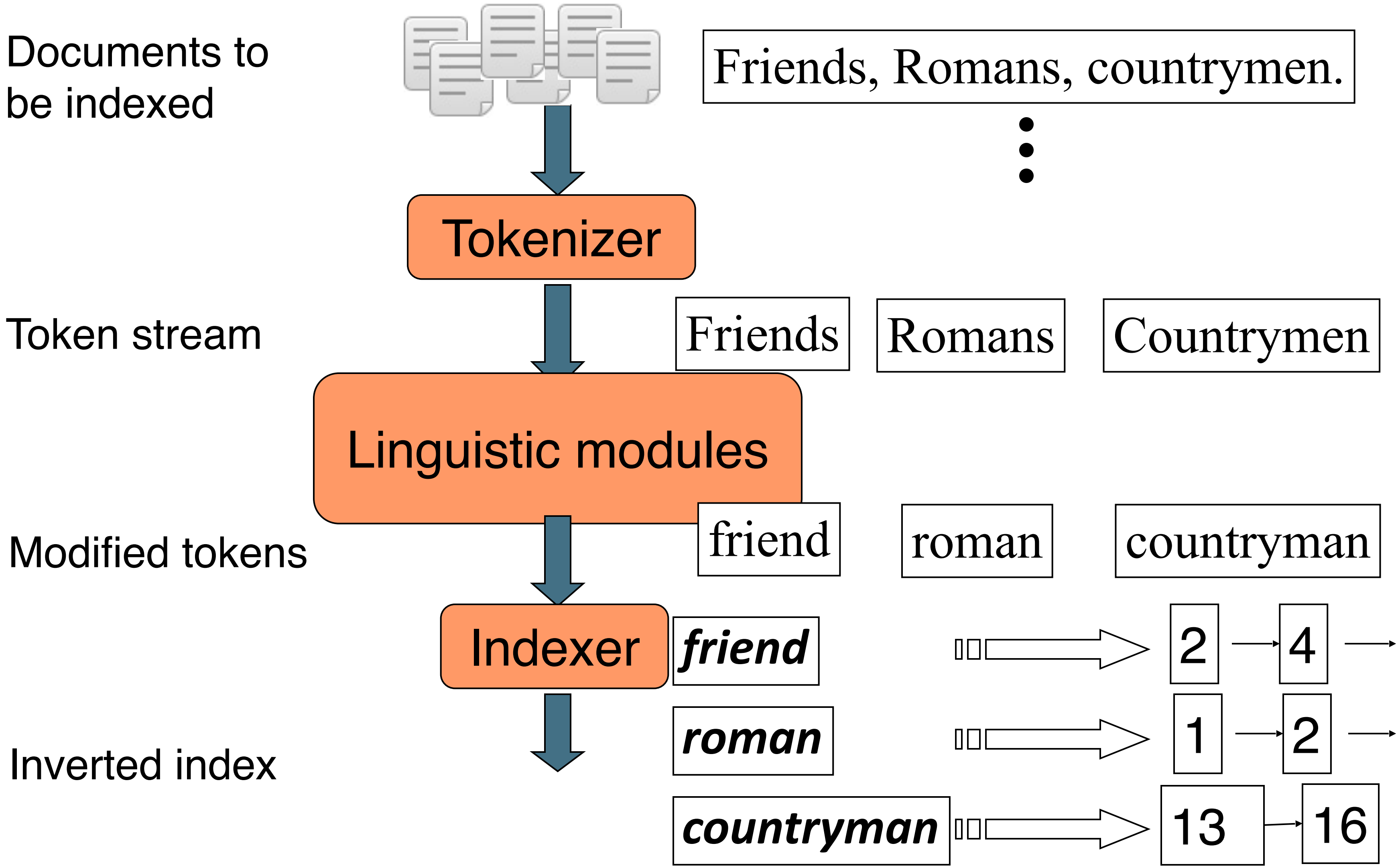
- We need variable-size postings lists
  - On disk, a continuous run of postings is normal and best
  - In memory, can use linked lists or variable length arrays
    - Some tradeoffs in size/ease of insertion (what's the insertion complexity?)



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

Sorted by docID (more later on why).

Building inverted index



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

Text preprocessing for building inverted index

- 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 step: Tokenize sequence

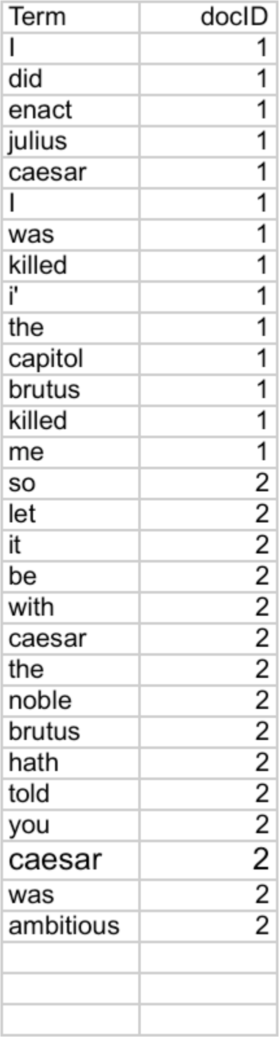
- Sequence of (Modified token, Document ID) pairs.

Doc 1

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

Doc 2

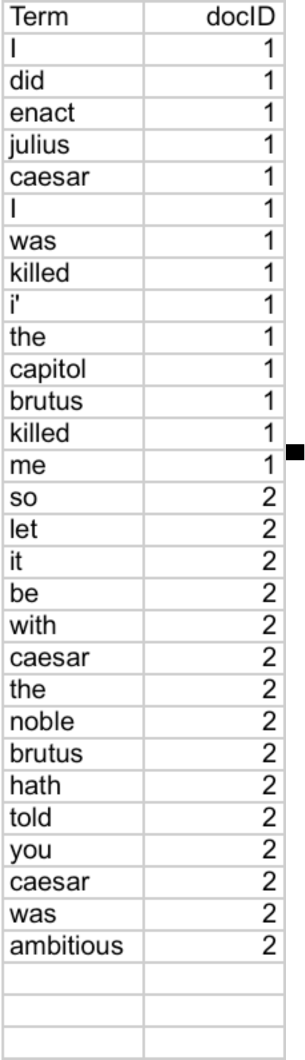
So let it be with
Caesar. The noble
Brutus hath told you
Caesar was ambitious



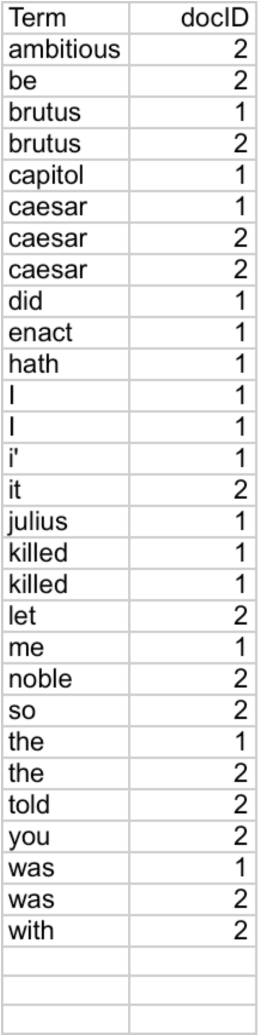
Term	docID
I	1
did	1
enact	1
julius	1
caesar	1
I	1
was	1
killed	1
i'	1
the	1
capitol	1
brutus	1
killed	1
me	1
so	2
let	2
it	2
be	2
with	2
caesar	2
the	2
noble	2
brutus	2
hath	2
told	2
you	2
caesar	2
was	2
ambitious	2

Indexer step: Sorting index terms

- First sort by term then by docID



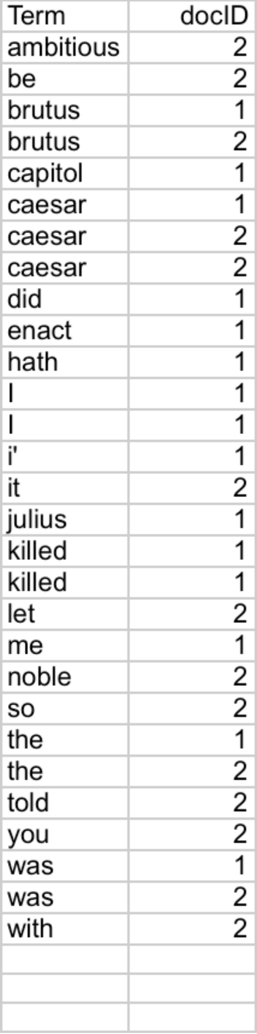
Term	docID
I	1
did	1
enact	1
julius	1
caesar	1
I	1
was	1
killed	1
i'	1
the	1
capitol	1
brutus	1
killed	1
me	1
so	2
let	2
it	2
be	2
with	2
caesar	2
the	2
noble	2
brutus	2
hath	2
told	2
you	2
caesar	2
was	2
ambitious	2



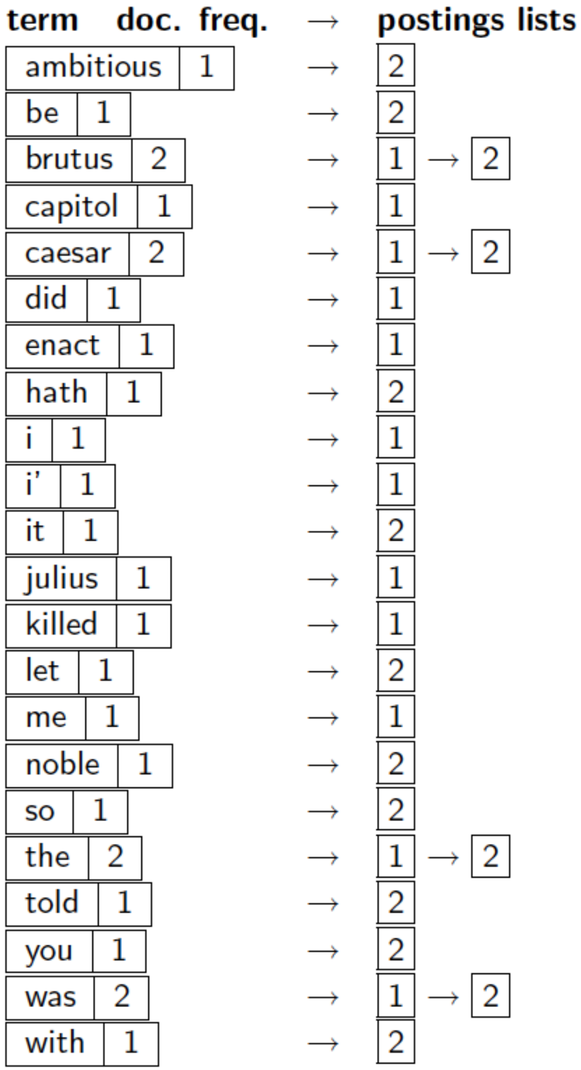
Term	docID
ambitious	2
be	2
brutus	1
brutus	2
capitol	1
caesar	1
caesar	2
caesar	2
did	1
enact	1
hath	1
I	1
I	1
i'	1
it	2
julius	1
killed	1
killed	1
let	2
me	1
noble	2
so	2
the	1
the	2
told	2
you	2
was	1
was	2
with	2

Indexer step: Dictionary and postings

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

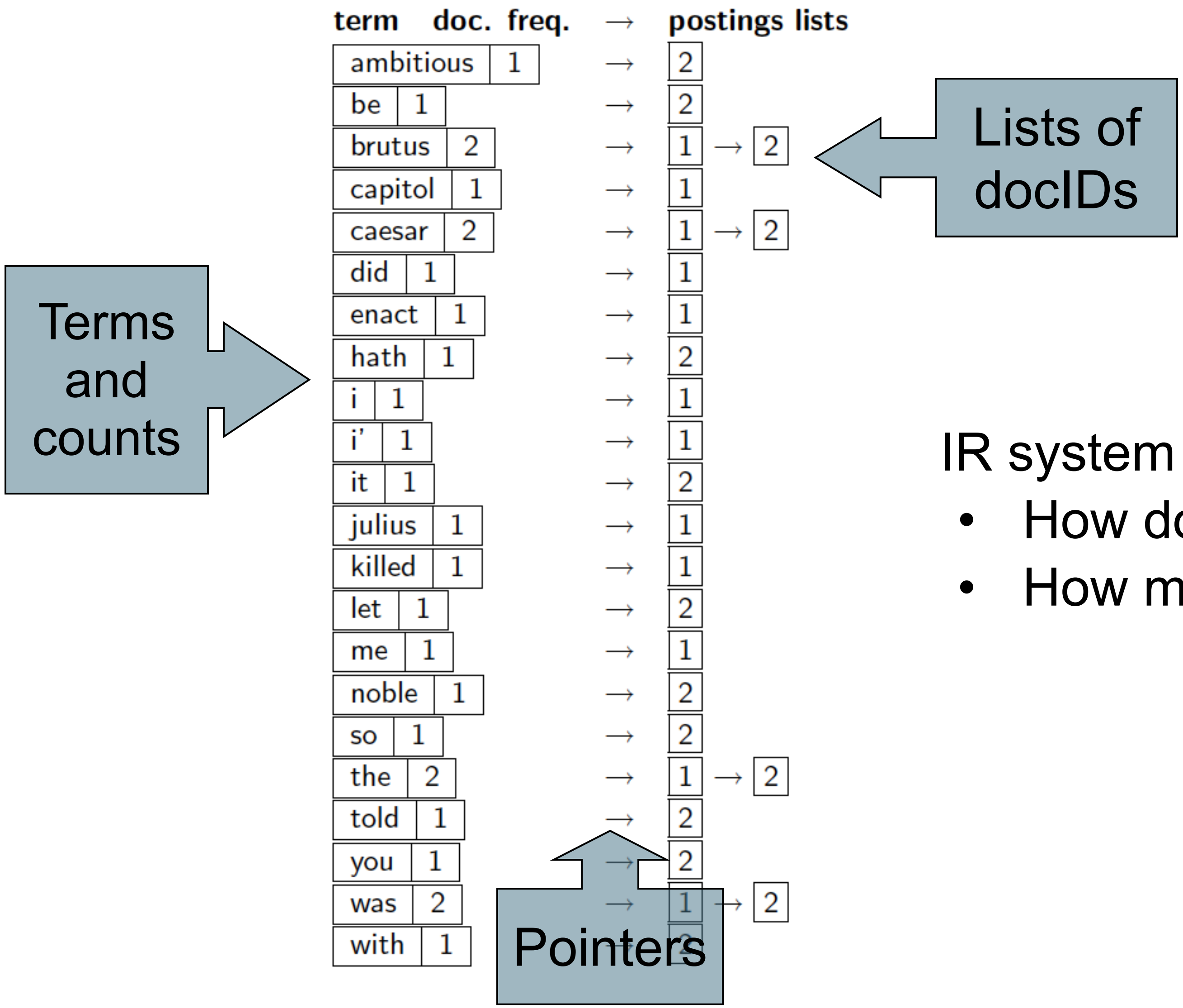


Term	docID
ambitious	2
be	2
brutus	1
brutus	2
capitol	1
caesar	1
caesar	2
caesar	2
did	1
enact	1
hath	1
I	1
I	1
i'	1
it	2
julius	1
killed	1
killed	1
let	2
me	1
noble	2
so	2
the	1
the	2
told	2
you	2
was	1
was	2
with	2



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

Terms and counts

Lists of docIDs

Pointers

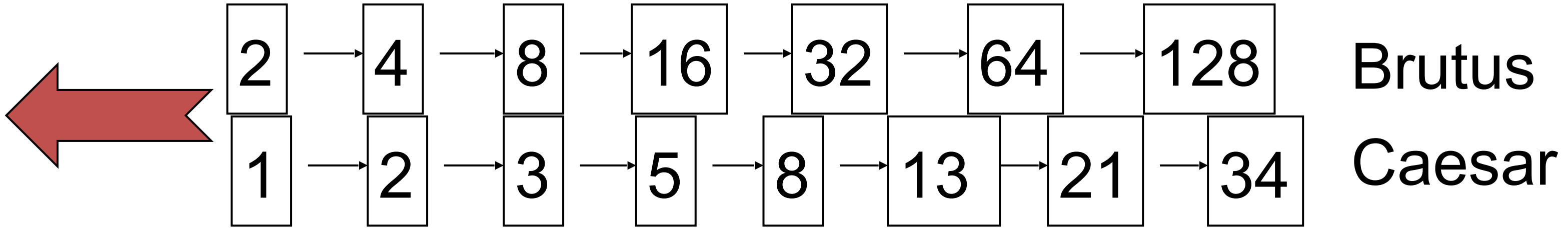
IR system implementation

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

Query processing for inverted index

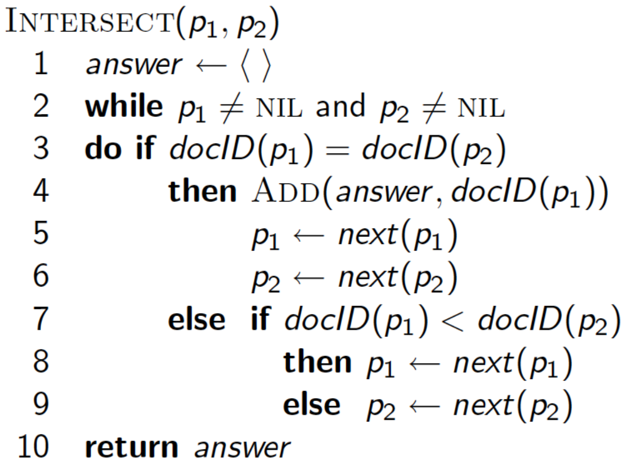
- Suppose we have constructed the inverted index, what query can we answer?

- 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):

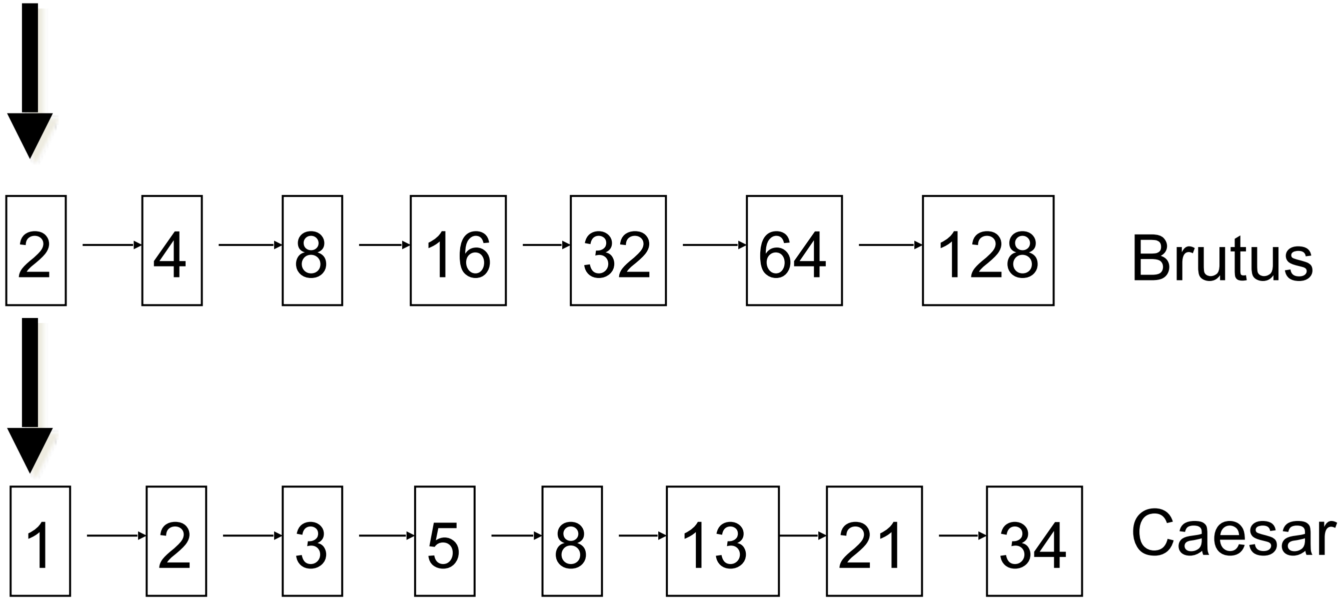


Merging two posting lists

- Walk through the two postings simultaneously, in time linear in the total number of postings entries (using two pointers, without skipping O(x + y))

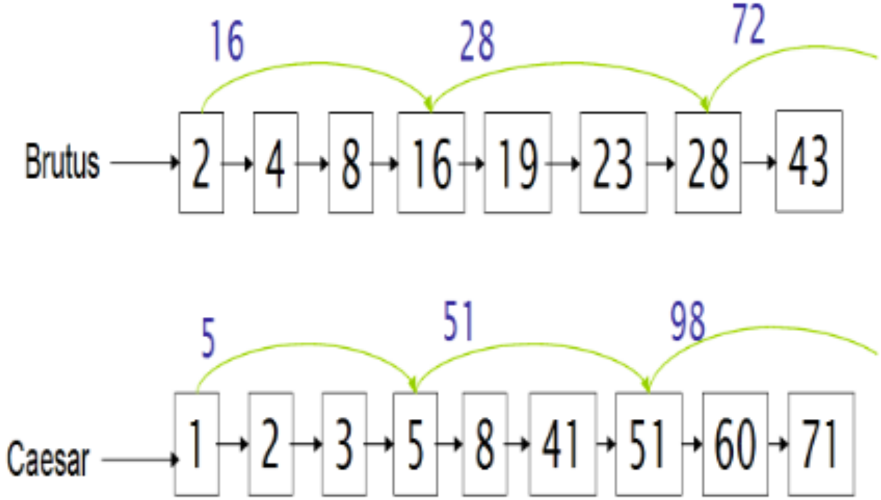


```
INTERSECT($p_1, p_2$)
answer ← ⟨ ⟩
while $p_1 \neq$ NIL and $p_2 \neq$ NIL
do if $docID(p_1) = docID(p_2)$
      then ADD($answer, docID(p_1)$)
           $p_1 \leftarrow next(p_1)$
           $p_2 \leftarrow next(p_2)$
      else if $docID(p_1) < docID(p_2)$
              then $p_1 \leftarrow next(p_1)$
              else $p_2 \leftarrow next(p_2)$
return answer
```



Merging two posting lists: skipping lists

- Speeding up the merge by skipping every k pointers



Inverted index for the Boolean Retrieval System

- 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, macOS 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?

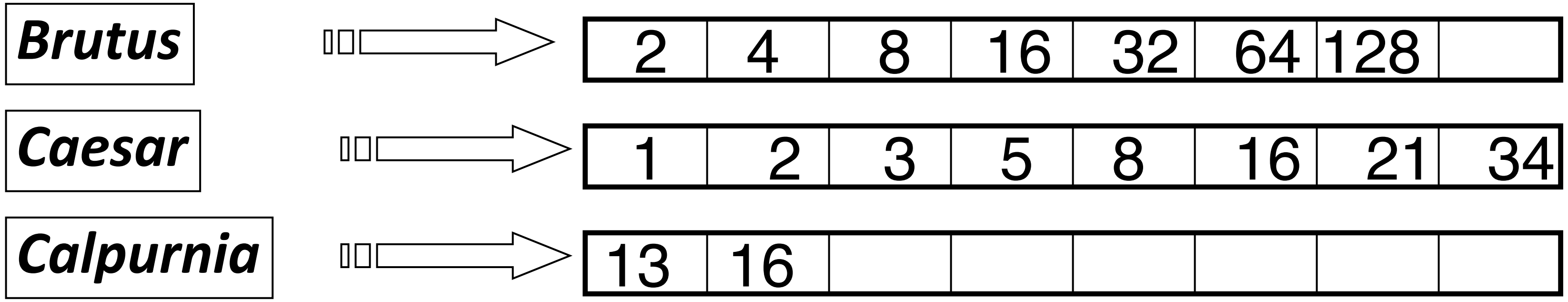
What about arbitrary Boolean formula?

(Brutus *OR* ***Caesar)*** *AND NOT*

(Antony *OR* ***Cleopatra)***

- Can we always merge in “linear” time?

- Consider a query that is an *AND* of *n* terms, what is the best way of processing?

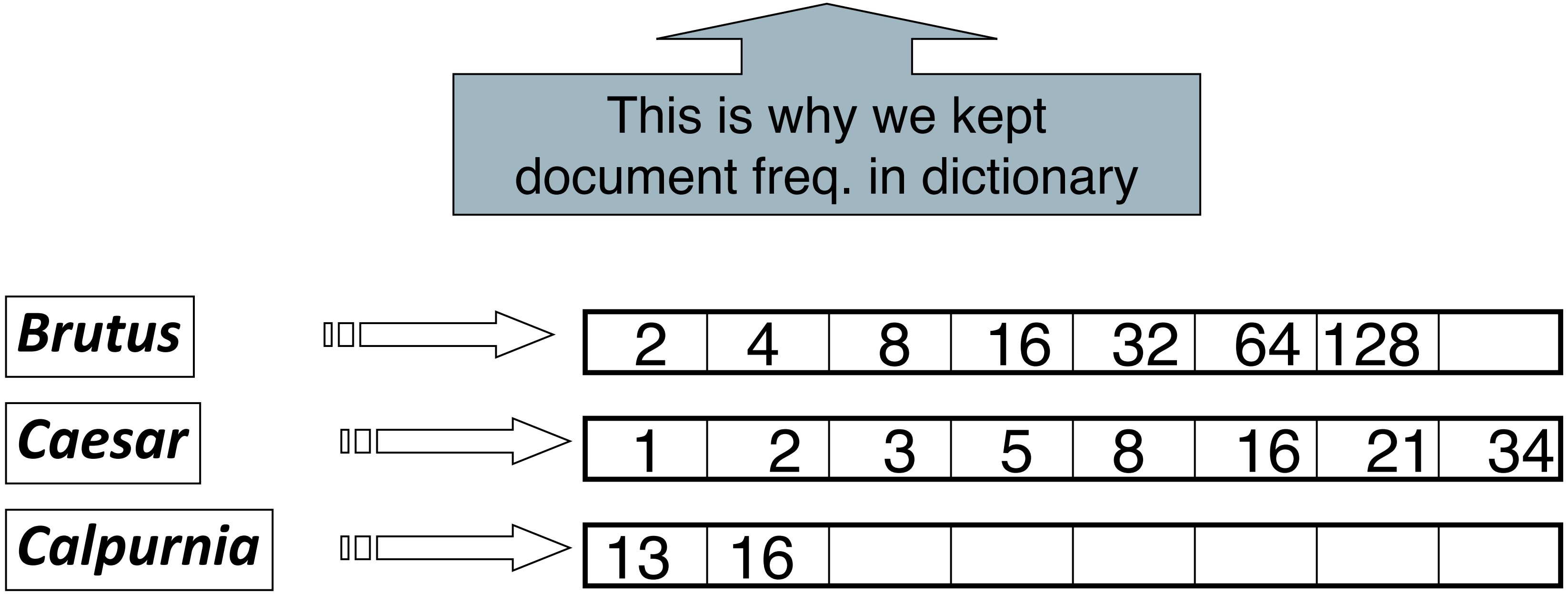


Brutus	2	4	8	16	32	64	128	
Caesar	1	2	3	5	8	16	21	34
Calpurnia	13	16						

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

What about arbitrary Boolean formula?

- Process in order of increasing freq:
  - *start with smallest set, then keep cutting further*.



This is why we kept document freq. in dictionary

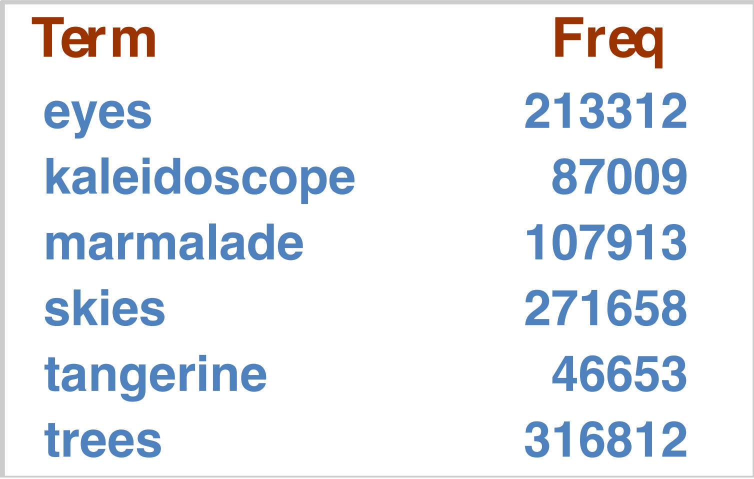
Brutus	2	4	8	16	32	64	128	
Caesar	1	2	3	5	8	16	21	34
Calpurnia	13	16						

Query: ***(Calpurnia*** AND ***Brutus***) *AND* ***Caesar***

Query processing 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

- Which terms should we process first?

Phrase indexing

- We want to be able to answer queries such as **“*Stevens Institute of Technology*”** – as a phrase

- Sentence 1= *“I went to **Stevens Institute** of **Technology**”*

- Sentence 2= “The **Technology Institute** that **Steve** Harvey went to. ”

- For this, it no longer suffices to store only
- <*term* : *docs*> entries

A first attempt: bi-gram indexing

- 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.

Issues with bi-gram indexing

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

Solution 2: Positional indexing

- 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.>

Solution 2: Positional indexing

<***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

Proximity search

- 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; …
- LIMIT! /3 to /3 be /2 or /2 not
    - Here, /*k* means "within *k* words of".

Positional indexing size

- A positional index expands postings storage *substantially*
  - Even though indices can be compressed

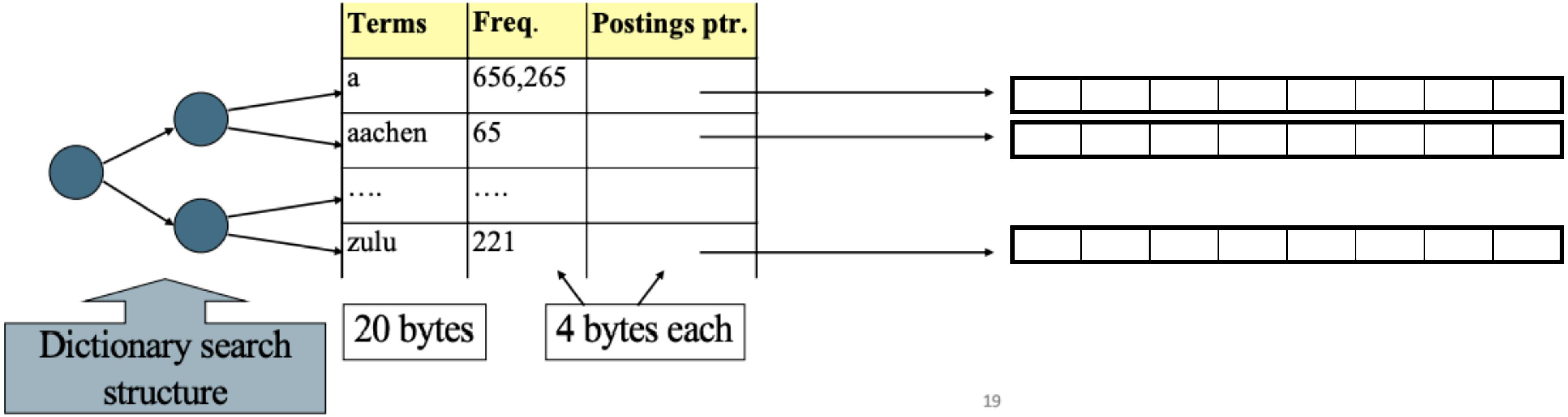
- Nevertheless, a positional index is now standardly used because of the power and usefulness of phrase and proximity queries … whether used explicitly or implicitly in a ranking retrieval system

- A positional index is 2–4 as large as a non-positional index

- Positional index size 35–50% of volume of original text

Index compression (for non-positional indexing)

- Compressing the posting pointer table
- Compressing the posting lists
- Speeding up the dictionary search with trie



Terms	Freq.	Postings ptr.
a	656,265	
aachen	65	
….	….	
zulu	221	

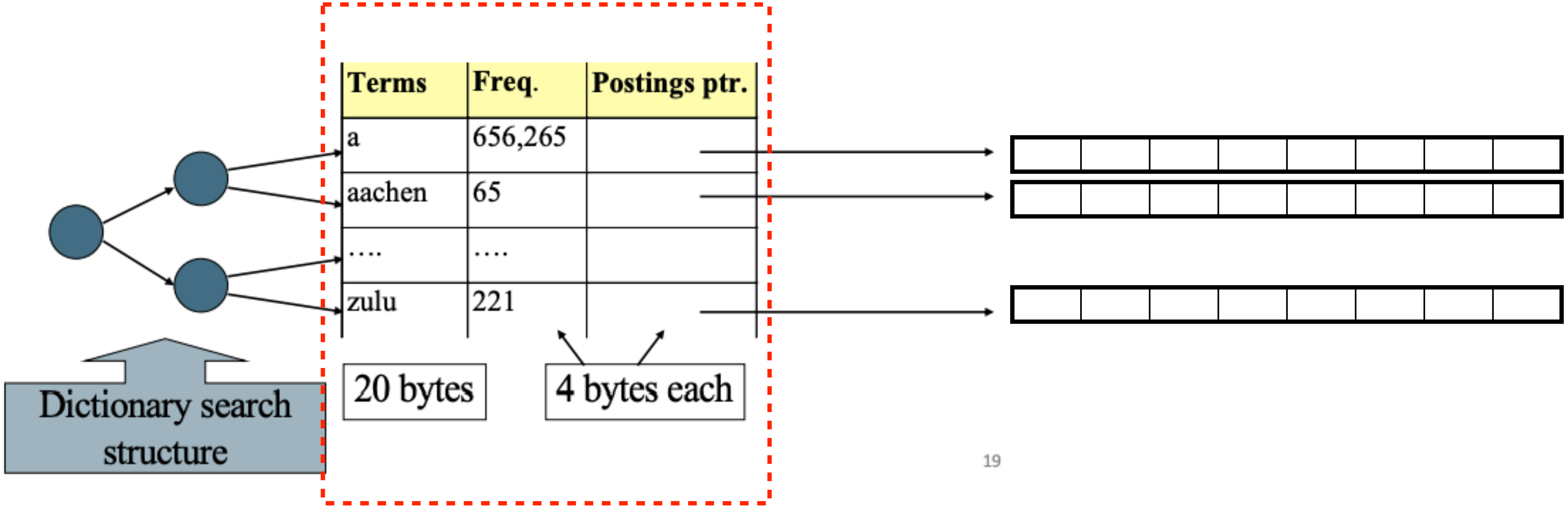
Speeding up the dictionary search

Compressing the posting ptr table

Compressing the posting list

Compressing the postings pointer table

- Most of the space in the table is wasted
  - Most words < 20 bytes
  - Table storage = 28N

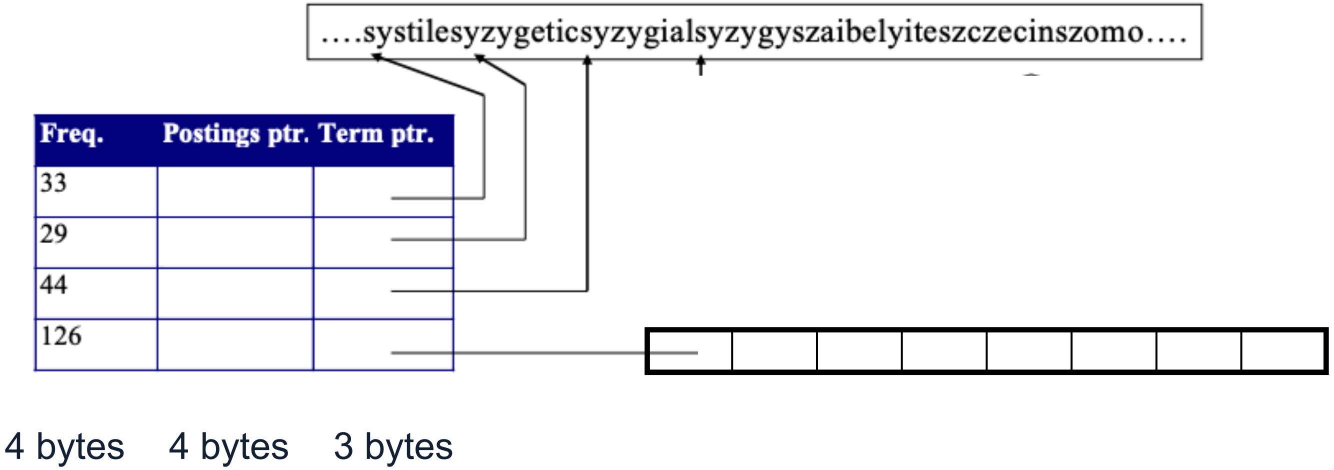


Terms	Freq.	Postings ptr.
a	656,265	
aachen	65	
….	….	
zulu	221	

Compressing the posting ptr table

Compressing the postings pointer table

- Concatenate the dictionary as one string
    - Table storage 28N -> 11N
- How to further improve the storage space?
    - Instead of storing absolute term pointers, store the gaps



….systilesyzygeticsyzygialsyzygyszaibelyiteszczecinszomo….

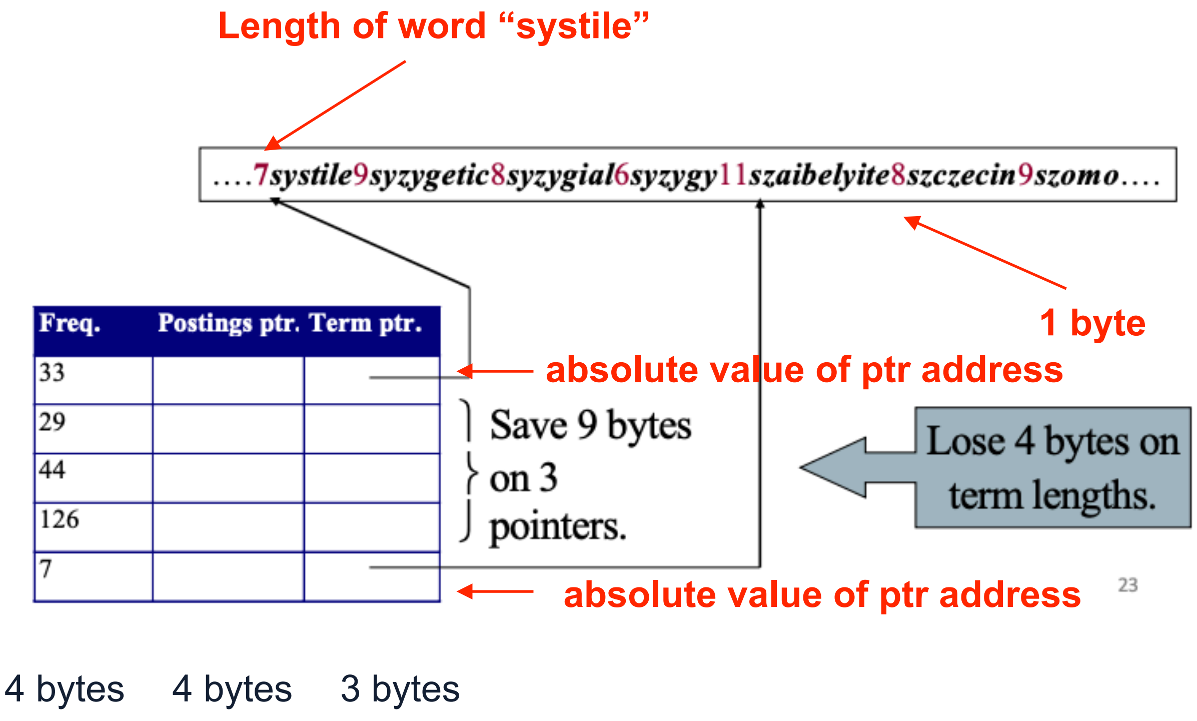
Freq.	Postings ptr.	Term ptr.
33		
29		
44		
126		
4 bytes	4 bytes	3 bytes

Compressing the postings pointer table

- Save more space by skipping (k-1) pointers for every k pointers
  - Recover the skipped pointers by adding the **length of words**

- Table storage is further reduced to 8N + 3N * (**3+k**/3*k). When k=4, the storage required is 9.75N < 11N

- Trade-off between saving space (skipping more) vs. saving time (skipping less)

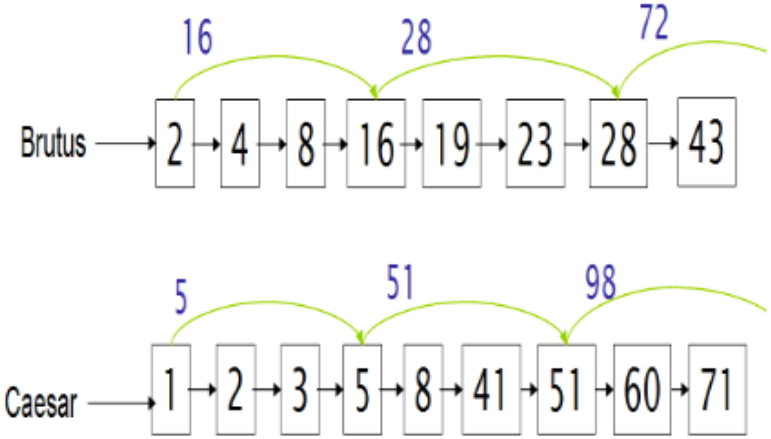


Length of word "systile"
….7systile9syzygetic8syzygial6syzygy11szaibelyite8szczecin9szomo….
1 byte
absolute value of ptr address
Save 9 bytes on 3 pointers.
Lose 4 bytes on term lengths.
absolute value of ptr address
23
4 bytes
4 bytes
3 bytes

Freq.	Postings ptr.	Term ptr.
33		
29		
44		
126		
7		

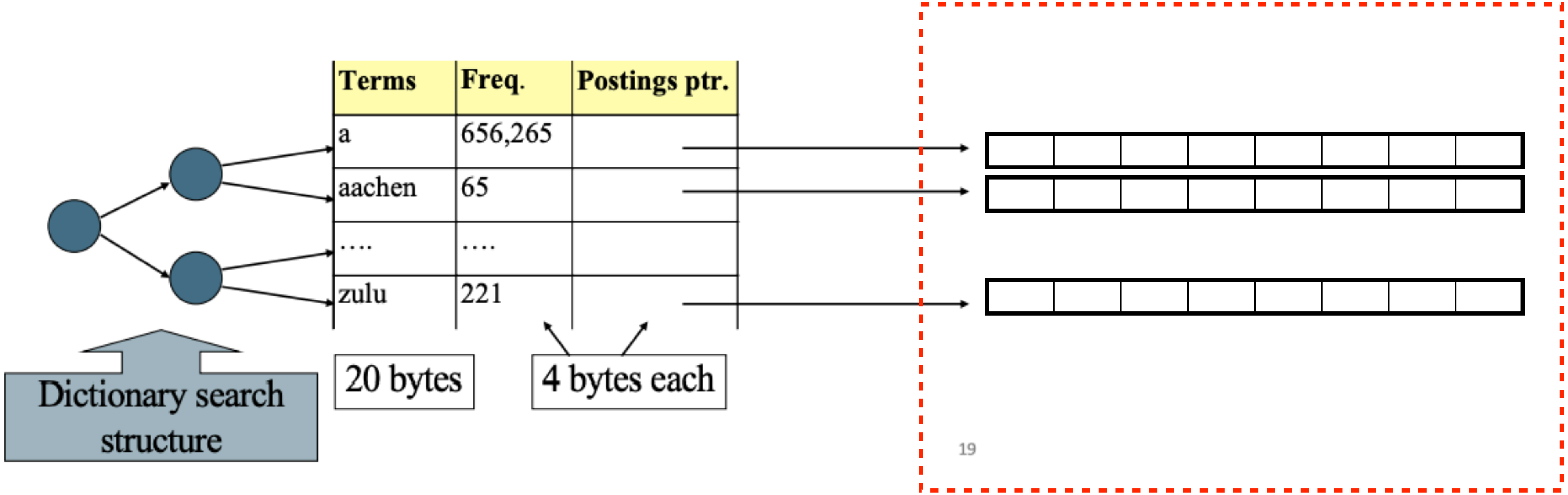
Compressing the posting lists

- Observations of posting files
  - Instead of storing docID, store gaps
  - Brutus: 2,4,8,3,4,5,15
  - Binary seq: 10,100,1000,11,100,101,1111



- Prefix encoding
  - Binary encoding such that the sequence can be uniquely decoded (why do we need this uniqueness?)
  - e.g., Huffman encoding
  - Unary encoding: {2:110,4:11110, …}
  - A uniquely decodable seq: 11011110111111101110…

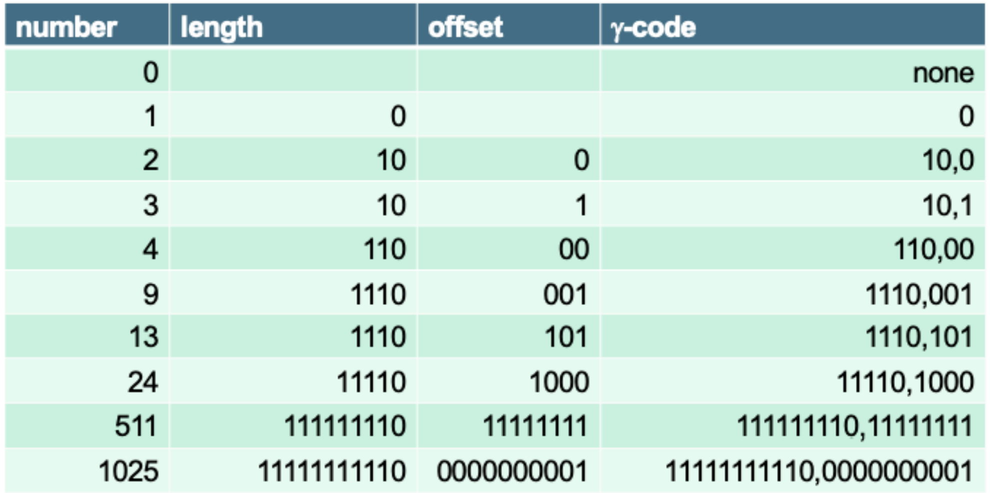
Compressing the posting lists



Terms	Freq.	Postings ptr.
a	656,265	
aachen	65	
….	….	
zulu	221	

Compressing the posting list

Compressing the posting lists



number	length	offset	γ-code
0			none
1	0		0
2	10	0	10,0
3	10	1	10,1
4	110	00	110,00
9	1110	001	1110,001
13	1110	101	1110,101
24	11110	1000	11110,1000
511	111111110	11111111	111111110,11111111
1025	11111111110	0000000001	11111111110,0000000001

- Unary encoding is too long!

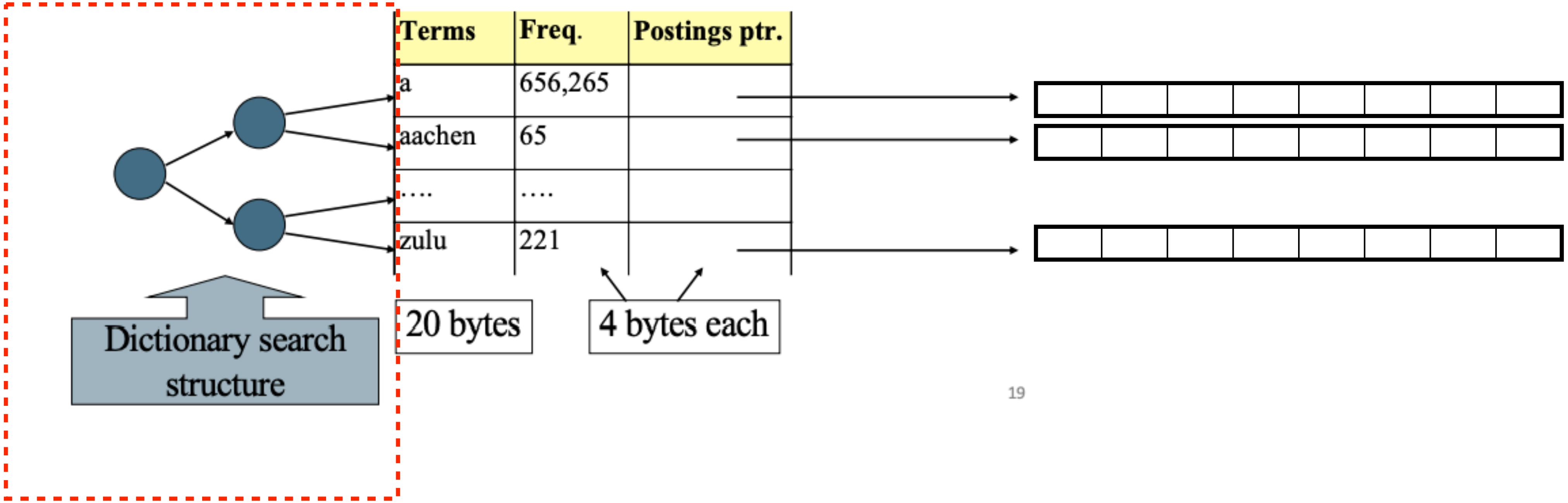
- **Gamma code** of 13: 1110,101
  - Binary code for {length - 1} followed by 0: 1110
  - Offset (last {length - 1} bits of the binary value): 13 =1101 → 101

- What is the gamma code of 5? 101 -> 110,01

- We can prove gamma code is uniquely decodable!

- Gamma code compression rate: 11.7%

Speeding up the dictionary search

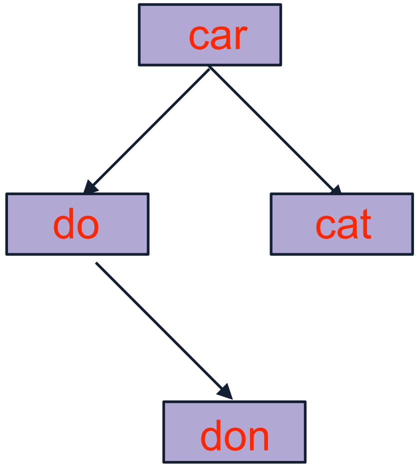


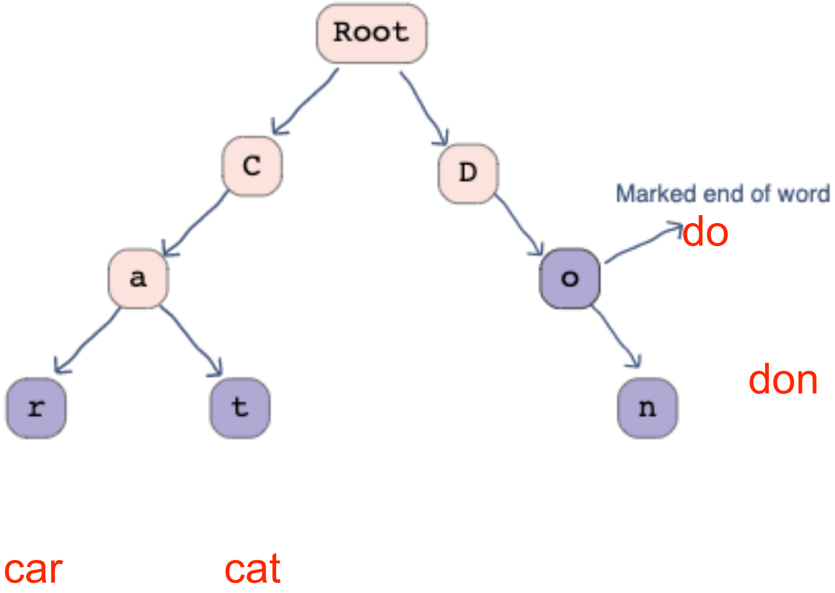
Terms	Freq.	Postings ptr.
a	656,265	
aachen	65	
….	….	
zulu	221	

Speeding up the dictionary search

Speeding up the dictionary search with prefix tree

- Time complexity for searching/insertion:
  - BST: O(m * log n), m is the maximum word length, n is the number of words in the vocabulary
  - Prefix-tree: O(m), m is the maximum word length





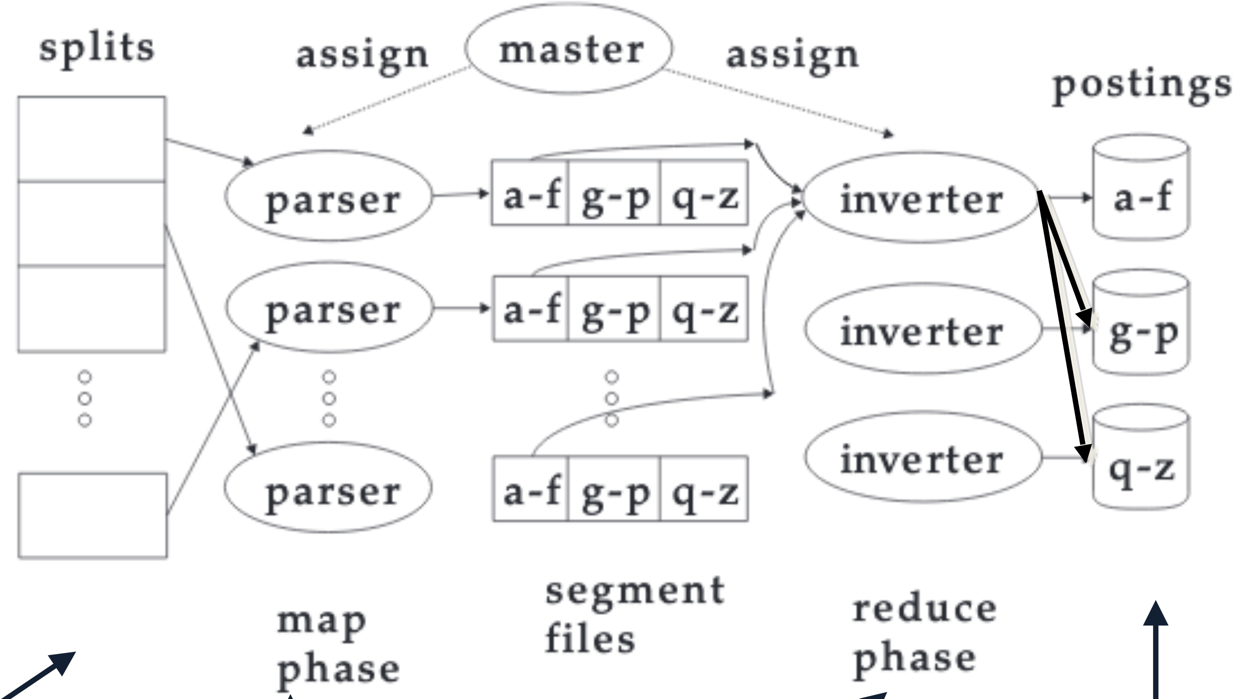
Handling web scale indexing

- Web-scale indexing must use clusters of servers
  - Google had 1 million servers in 2011

- Fault tolerance of a massive data center
  - If a non-fault tolerance system has 1000 nodes, each has 99.9% uptime, then 63% of the time one or more servers is down

- Solution
  - Maintain a “master” server
  - Break indexing into parallel tasks
  - Assign each task to an idle machine

Map-reduce



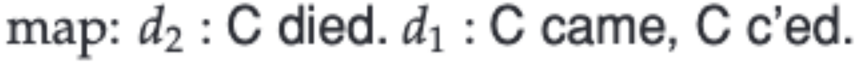
master assigns split to idle machine

parser emits (term,doc) pair

merge partitions in inverter

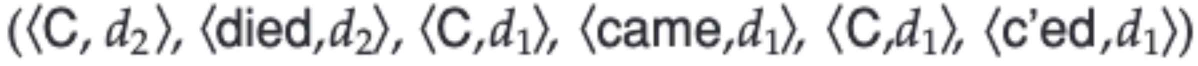
complete the index

Examples of map-reduce

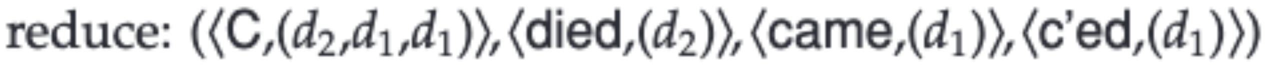


map: d_2 : C died. d_1 : C came, C c'ed.

→

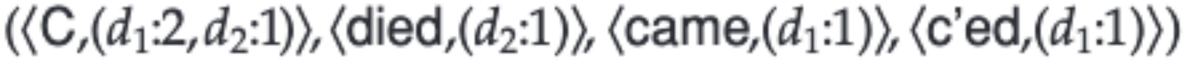


(⟨C, d_2⟩, ⟨died,d_2⟩, ⟨C,d_1⟩, ⟨came,d_1⟩, ⟨C,d_1⟩, ⟨c'ed,d_1⟩)



reduce: (⟨C,(d_2,d_1,d_1)⟩,⟨died,(d_2)⟩,⟨came,(d_1)⟩,⟨c'ed,(d_1)⟩)

→

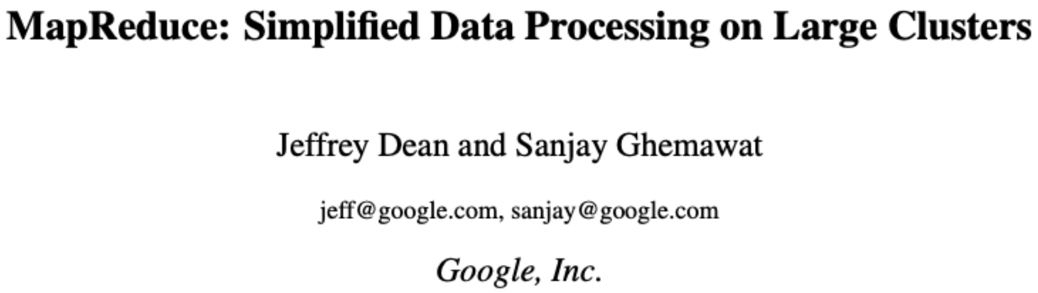


(⟨C,(d_1:2,d_2:1)⟩,⟨died,(d_2:1)⟩, ⟨came,(d_1:1)⟩,⟨c'ed,(d_1:1)⟩)

MapReduce: Industry practice

- Term partition vs. document partition
  - Term-partitioned: one machine handles a subrange of terms
  - Document-partitioned: one machine handles a subrange of documents

- Most industry search engine use document-partitioned index
  - Better load balancing (**why?**)



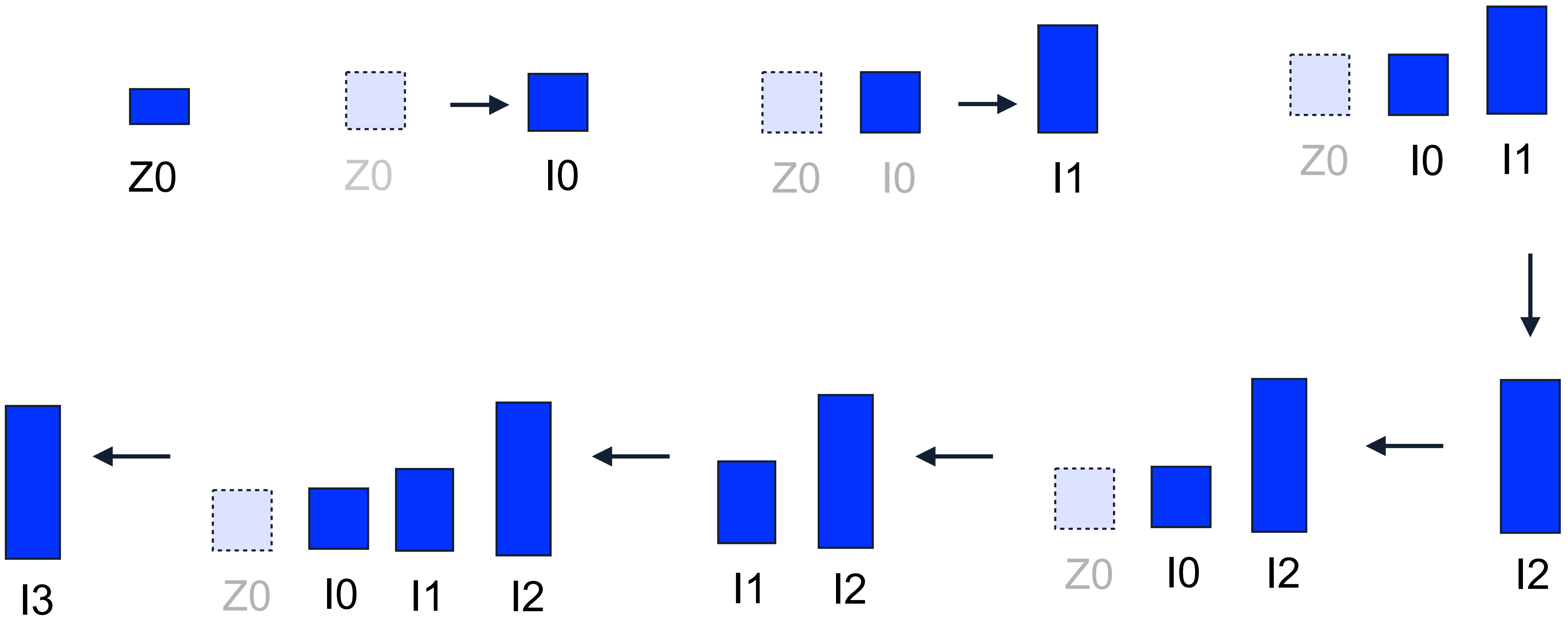
MapReduce: Simplified Data Processing on Large Clusters

Jeffrey Dean and Sanjay Ghemawat

jeff@google.com, sanjay@google.com

Google, Inc.

Logarithmic dynamic indexing



https://nlp.stanford.edu/IR-book/html/htmledition/dynamic-indexing-1.html

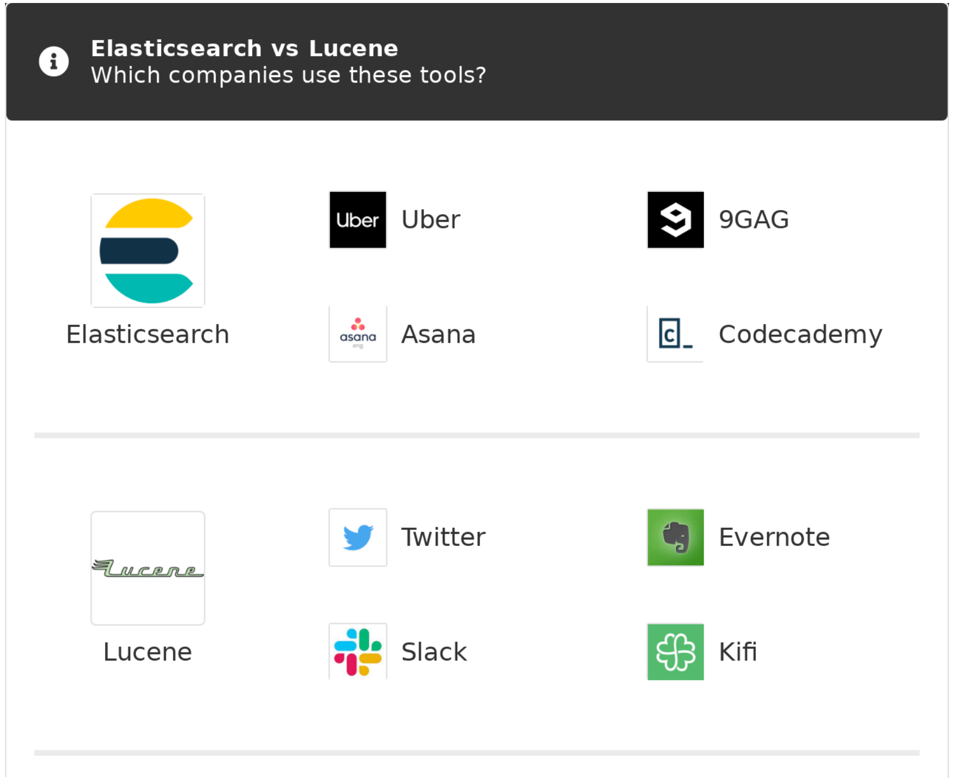
Real time search of Twitter

- Requires high real time search
  - Low latency, high throughput query evaluation
  - High ingestion rate and immediate data availability
  - Concurrent reads and writes of the index

- Solution: using segments
  - Each segment consists of 2^32 tweets (in memory)
  - New posts are appended to the posting lists
  - Only one segment can be written to at each time

Search engine tools

- Apache Lucene
  - Free and open search engine library
  - First developed in 1999
- ElasticSearch
  - A search engine
  - based on Lucene

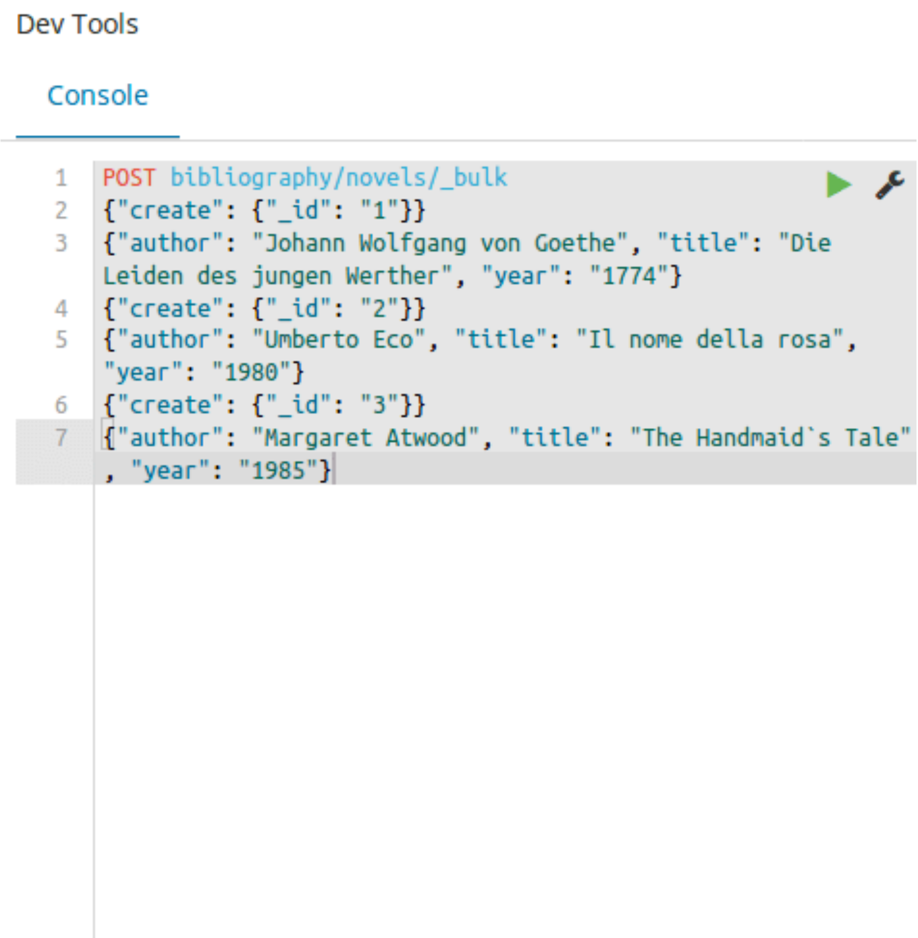


Elasticsearch vs Lucene
Which companies use these tools?

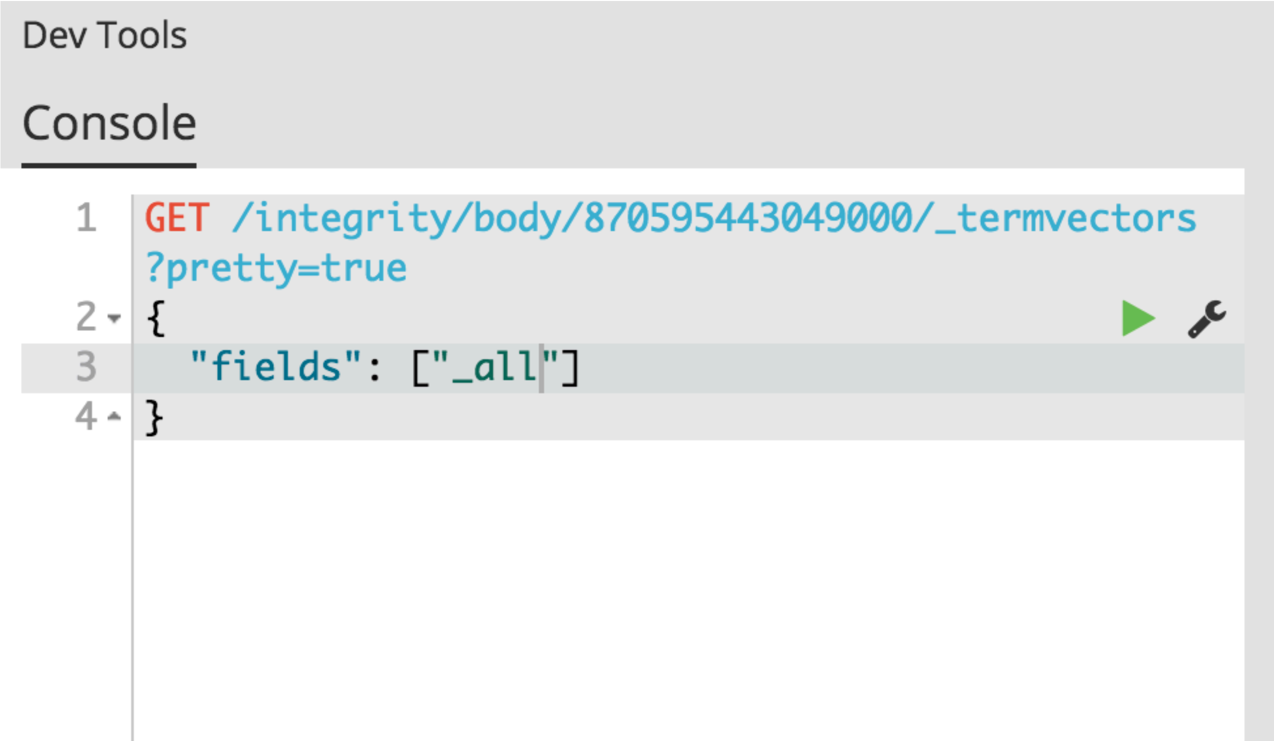
Elasticsearch	Uber	9GAG
	Asana	Codecademy
Lucene	Twitter	Evernote
	Slack	Kifi

ElasticSearch

- Using a REST api



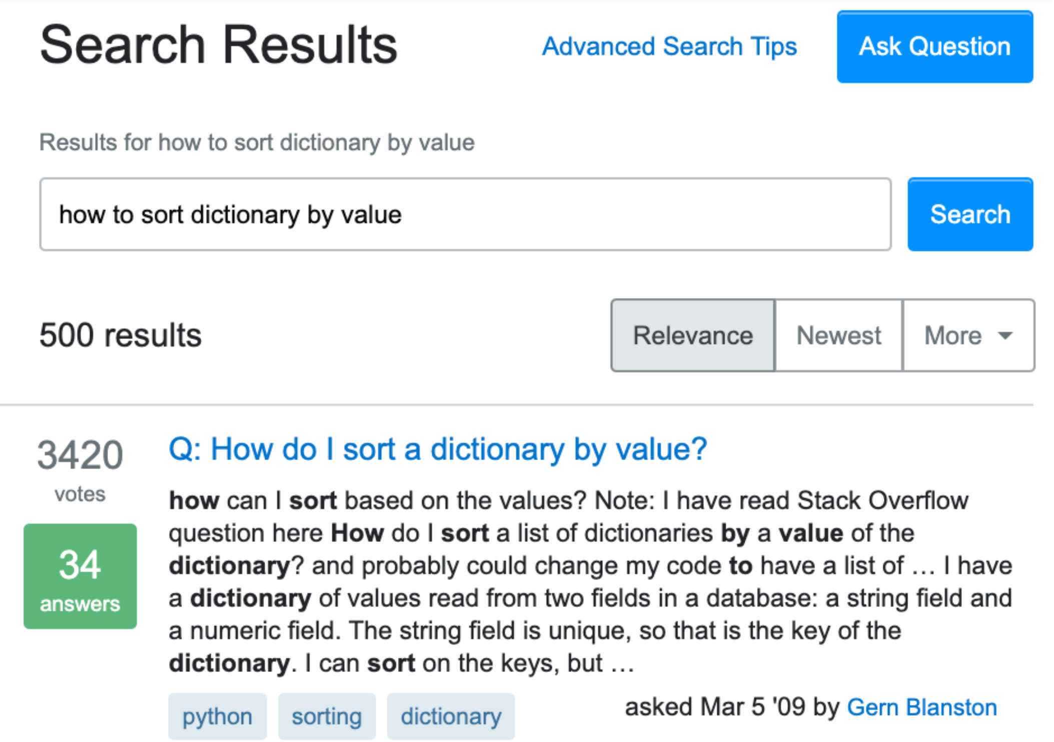
```
POST bibliography/novels/_bulk
{"create": {"_id": "1"}}
{"author": "Johann Wolfgang von Goethe", "title": "Die Leiden des jungen Werther", "year": "1774"}
{"create": {"_id": "2"}}
{"author": "Umberto Eco", "title": "Il nome della rosa", "year": "1980"}
{"create": {"_id": "3"}}
{"author": "Margaret Atwood", "title": "The Handmaid`s Tale", "year": "1985"}
```



```
GET /integrity/body/870595443049000/_termvectors?pretty=true
{
  "fields": ["_all"]
}
```

Homework 2: Using ElasticSearch to build a search engine

- Build an inverted index
- Evaluate three search algorithm's performance
    - TF-IDF
    - BM25
    - Dirichlet-LM



Search Results

Results for how to sort dictionary by value

how to sort dictionary by value

500 results

3420 votes

34 answers

Q: How do I sort a dictionary by value?

how can I **sort** based on the values? Note: I have read Stack Overflow question here **How** do I **sort** a list of dictionaries **by** a **value** of the **dictionary**? and probably could change my code **to** have a list of … I have a **dictionary** of values read from two fields in a database: a string field and a numeric field. The string field is unique, so that is the key of the **dictionary**. I can **sort** on the keys, but …

asked Mar 5 '09 by Gern Blanston