NLP4IS

- Welcome!
- This week (Feb 12th): **Information Retrieval**
  - Search in text collections: Boolean, inverted indices, ranked retrieval, vector space model, …
  - Basic things / classic concepts of IR
- Next unit (Feb 12th): **IR (VSM, ..), Word Embeddings**
  - Language models, word embeddings, word similarity, analogical reasoning, practical applications
  - Closer to current research / state of the art

About me

- Dr. Gerhard Wohlgenannt
- .. from Austria
- Working at ITMO since May 2017
- Before: Business University Vienna / Austria
  as Assistant Professor for 5 years
- **Contact:** gwohlg@corp.ifmo.ru

- **Everybody please start Pycharm** – cause it needs a lot of time to startup / index when started the first time

Overview of the first of my units

- **Unit 1:** IR
- **Unit 2:** IR, word embeddings
- **Unit 3:** word embeddings
- **Unit 4:** research related content of word embeddings, related to **projects**

  Projects:
  - 4-6 students with me (rest: Liubov, Ivan)
  - Little **clearly defined tasks**, with the goal to write a small scientific paper – **published at arxiv.org**
  - The tasks are based on existing research in the field.

Introduction to

Information Retrieval

Introducing Information Retrieval and Web Search

Most of content by: Prof Chris Manning, Stanford university

Reading .. For more information

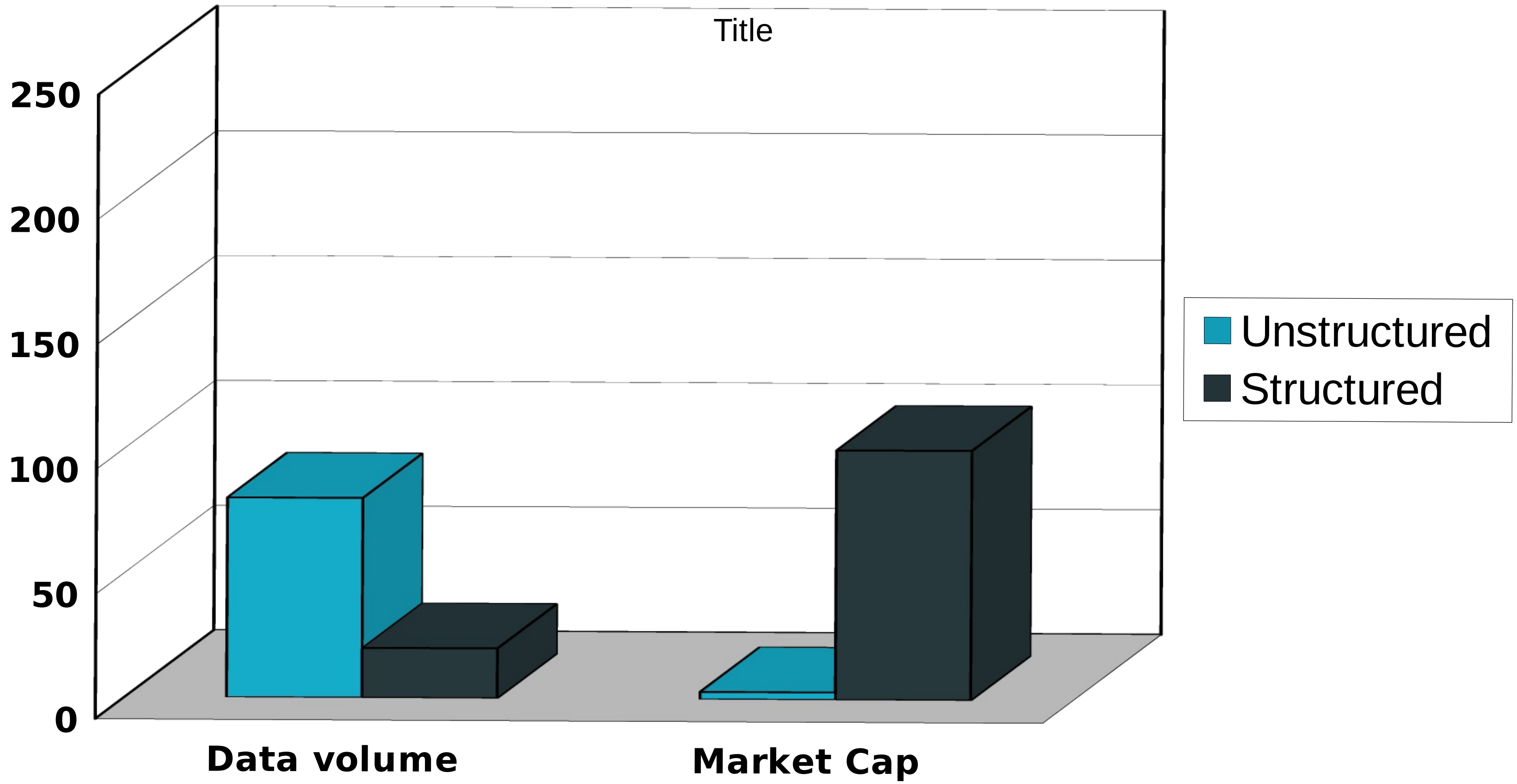
- https://nlp.stanford.edu/IR-book/
- http://www.nltk.org/book/

Information Retrieval

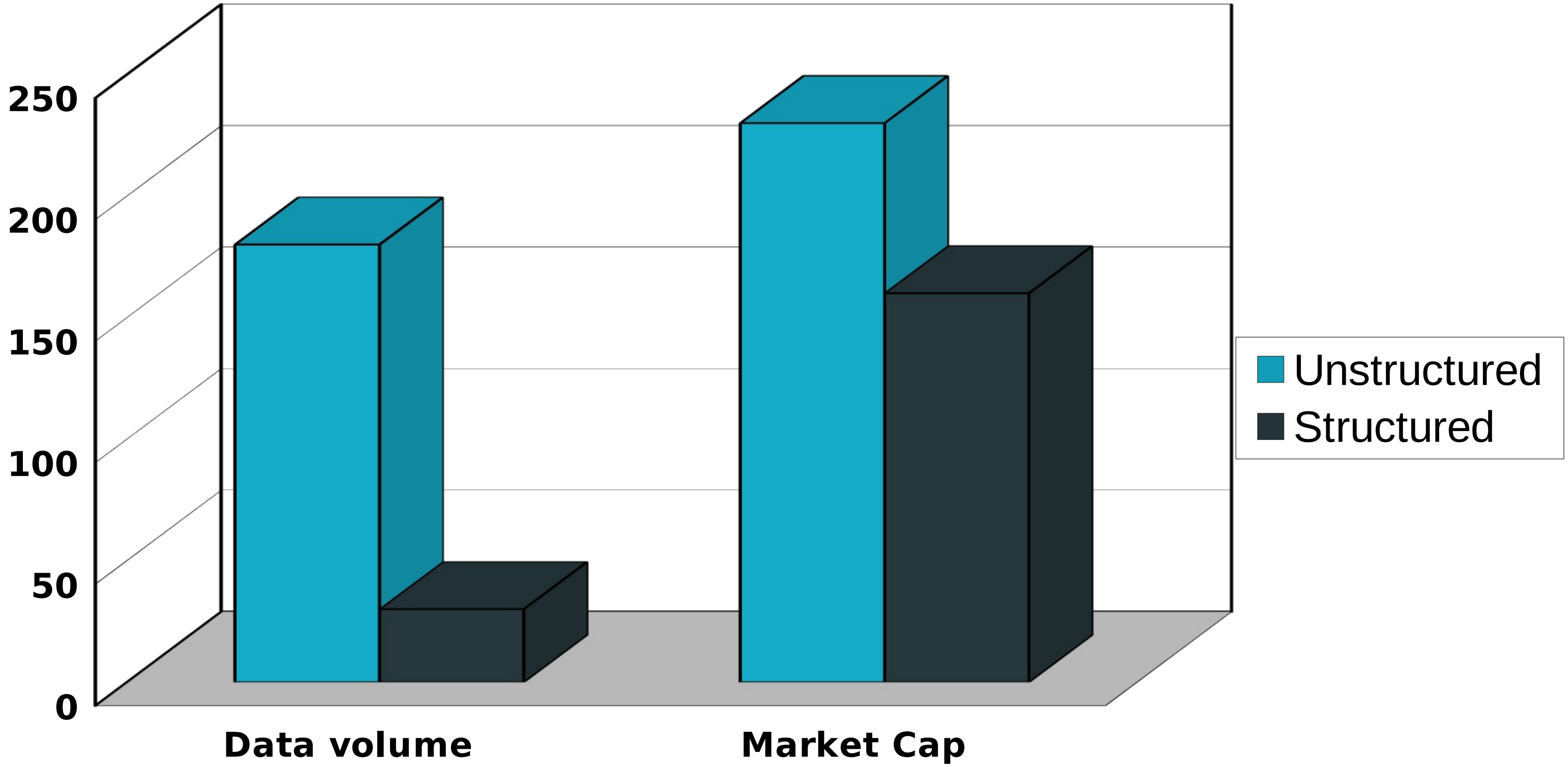
- Information Retrieval (IR) is finding material (usually documents) of an unstructured nature (usually text) that satisfies an information need from within large collections (usually stored on computers).

  - 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

Unstructured (text) vs. structured (database) data in the mid-nineties



Unstructured (text) vs. structured (database) data today

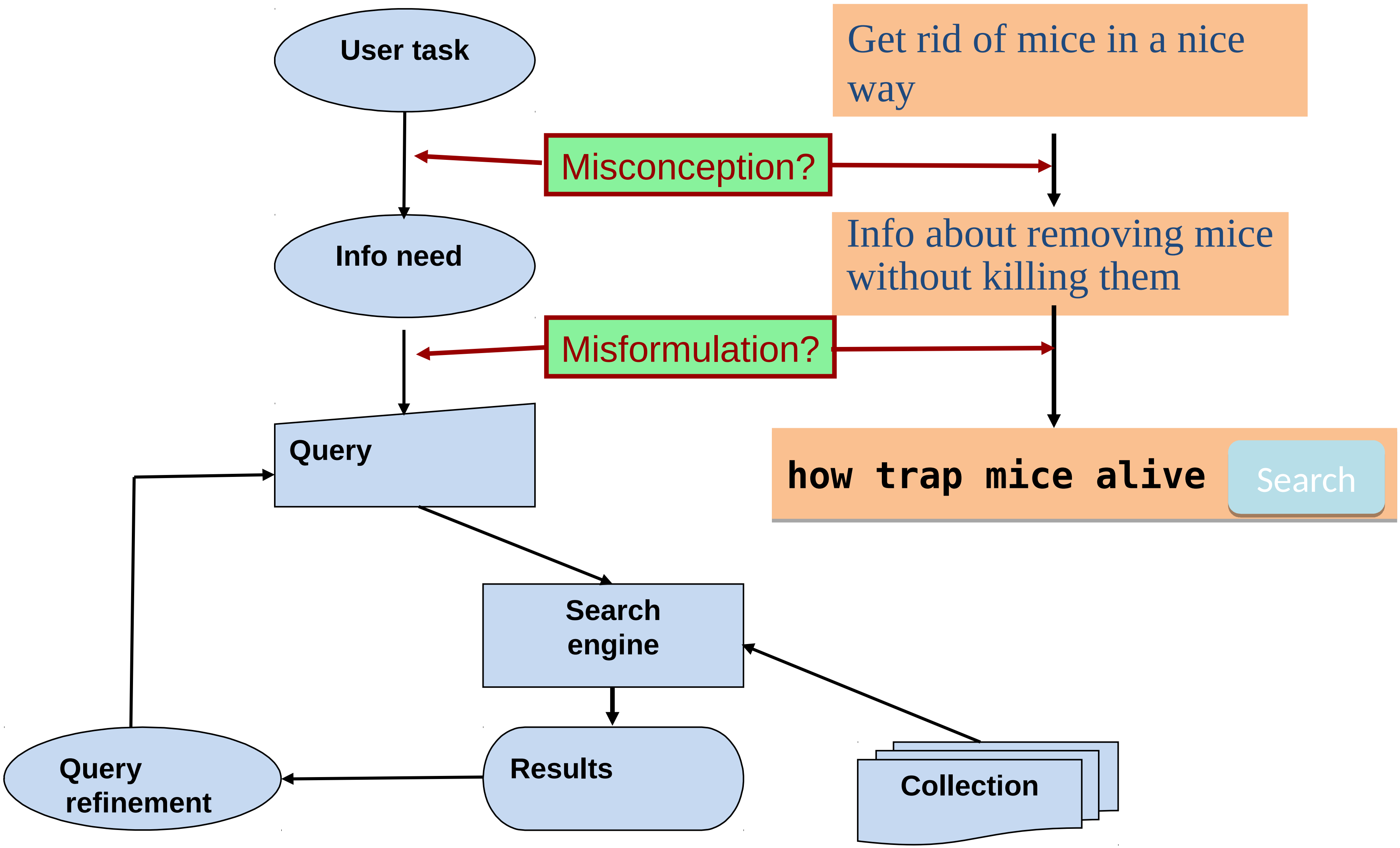


Basic assumptions of Information Retrieval

- 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

The classic search model



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

Introduction to

Information Retrieval

Term-document incidence matrices

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)
  - Other operations (e.g., find the word ***Romans*** near ***countrymen***) not feasible
  - Ranked retrieval (best documents to return)
    - Later lectures

First Code Example

- Basic Python
  - Who knows basic Python / who doesn't?
- Python easy to read and understand
  - For learning NLP with NTLK and Python at the same time: **http://www.nltk.org/book/**
- Basic NLTK: Corpora
  - `from nltk.book import * # 10 texts`
  - `text7.tokens[:100]`
  - `nltk.corpus.gutenberg # a collection of books`

Python NLTK Basics

- text1.concordance("monstrous")
- text4.dispersion_plot(["citizens", "democracy", "freedom", "duties", "America"])
- fdist1 = FreqDist(text1); fdist1.most_common(50)
- ..

First mini exercises (1)

First mini exercise:

- Run code1a.py and see what is does ..
- Play with the code, simple things: exchange the terms to be searched for, change parameters → just understand the code

First mini exercises (2)

First mini exercise:

- Run code1b.py and see what is does ..
- `find_in_gutenberg1()` searches for words (AND combined) in the texts
  - Understand the code
  - → extend it to do OR (with a parameter)

Problem here: too slow for interactive systems

One solution: Term-document matrix – see next slides

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

Exercise

- Show: code2_term_document_matrix.simple.py
- Implement: OR, NOT

• Bonus: handle strings like “a AND (b OR C) NOT d”

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.
- Simple incidence matrix not feasible any more

- But for small collections better implementations, like the TDM in code3_term_document_matrix.adv.py with pandas could be used

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.

Why?

- What's a better representation?
  - We only record the 1 positions.
  - **How can we do that??**

Introduction to

Information Retrieval

The Inverted Index

The key data structure underlying modern IR

Inverted index

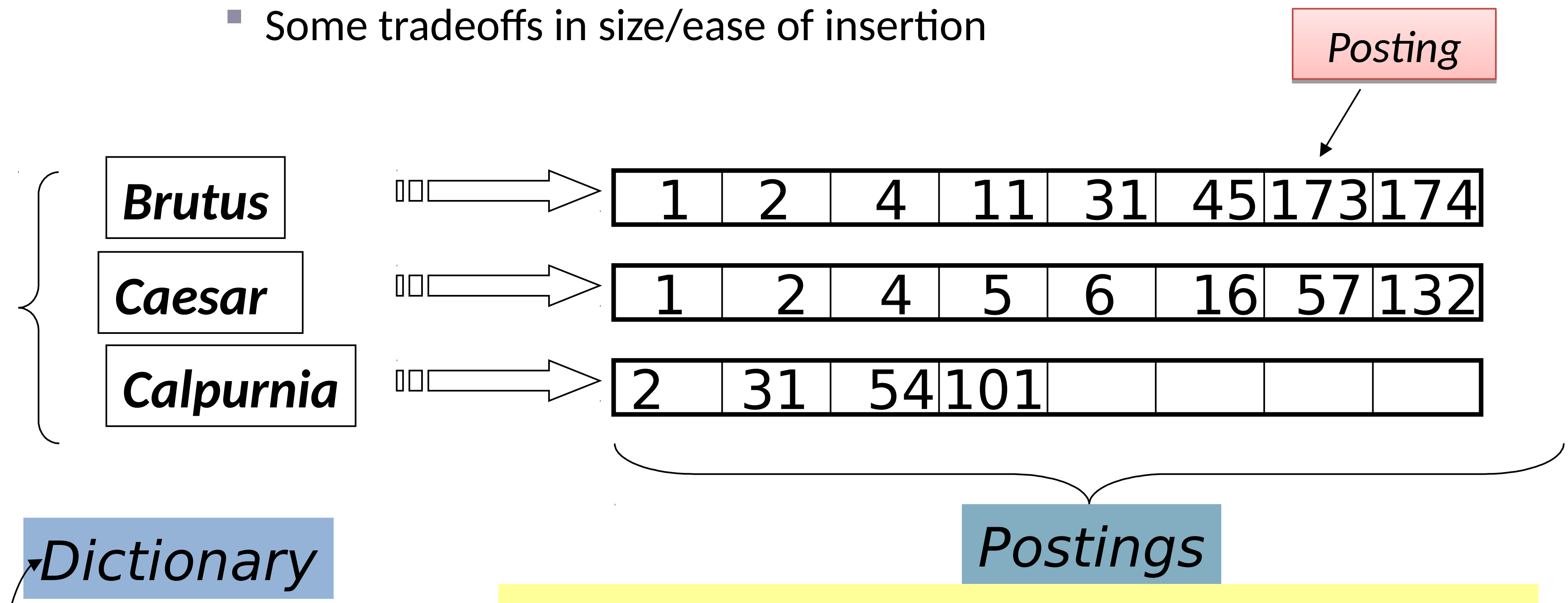
- For each term *t*, we must store a list of all documents that contain *t*.
  - Identify each doc by a **docID**, a document serial number
- Can we used fixed-size arrays for this?

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

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

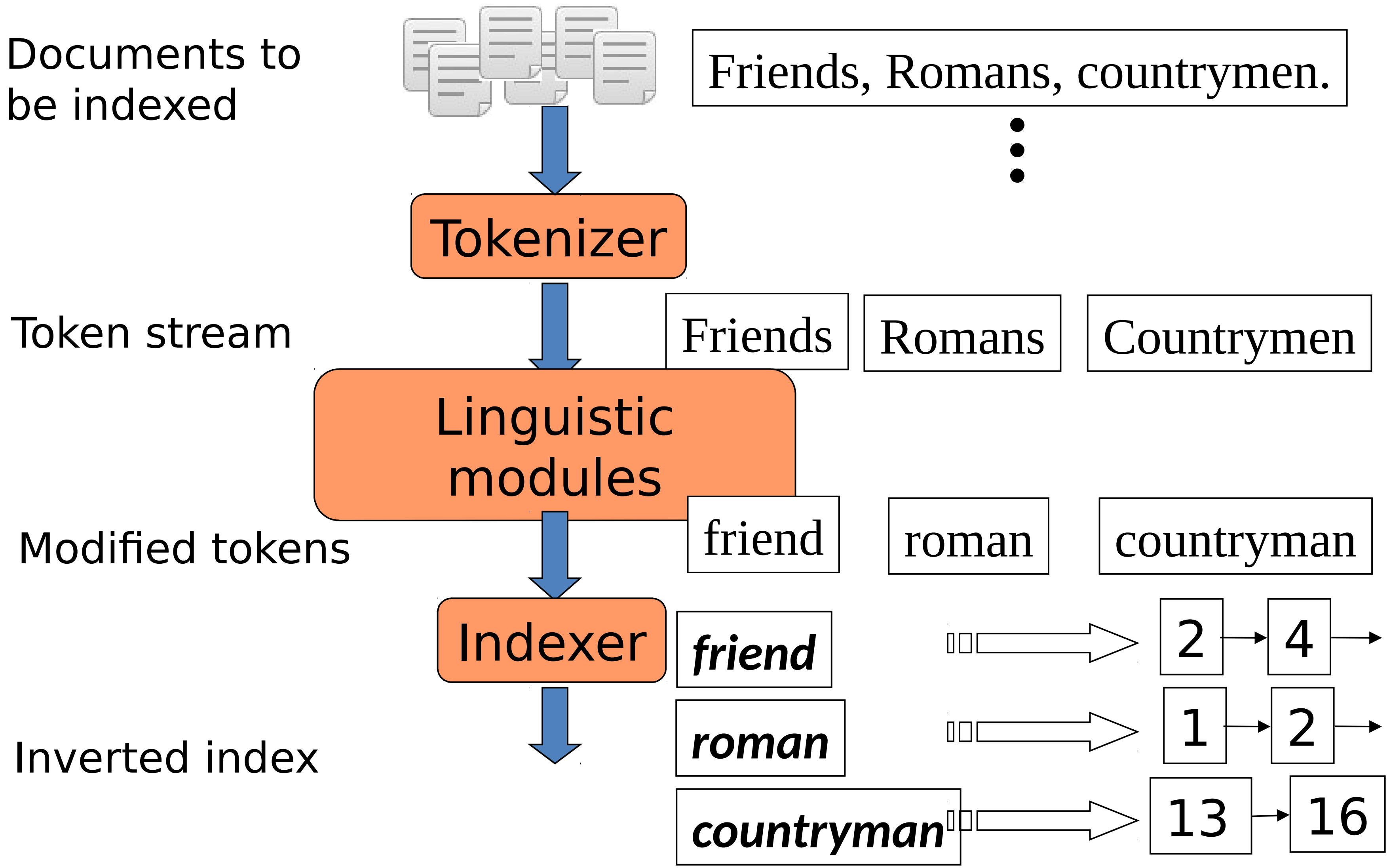
Inverted index

- 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



Sorted by docID (more later on why).

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***

Tokenization (in NLTK)

- In NLTK:
  - `from nltk.tokenize import sent_tokenize`
  - `from nltk.tokenize import word_tokenize`

    `# input to those is just a string`

- **Show code4 file (incl output)**
- Show Tokenizers in NLTK

Preprocessing .. code

- Show preprocess.py

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?

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?

Big Exercise

- Create an inverted index step-by-step like in the way shown above – for example for the gutenberg documents
  - You can limit the number of words per document to make it faster

---

- When done: Add **preprocessing** .. like in preprocess.py
  - What are the **effects** of preprocessing?
    - Number of words in index?
    - Cumulative length of posting lists?

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?

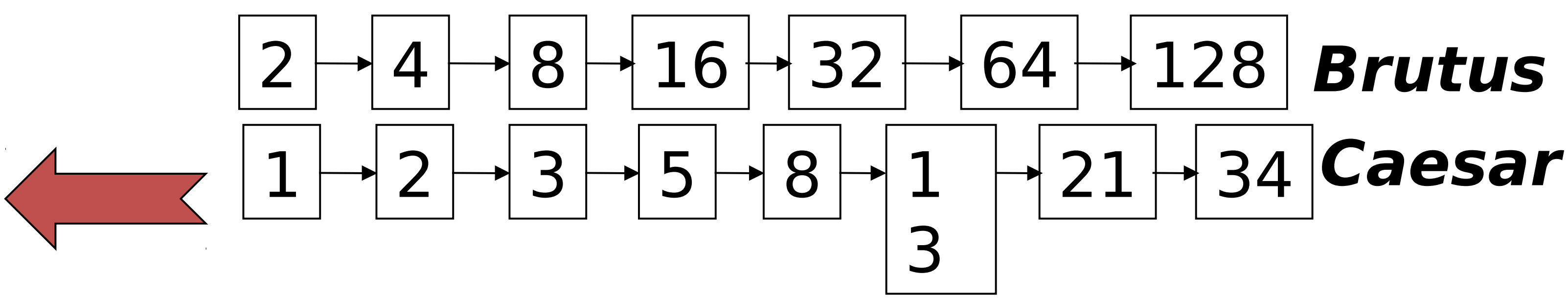
Our focus

Query processing: AND

- Consider processing the query:

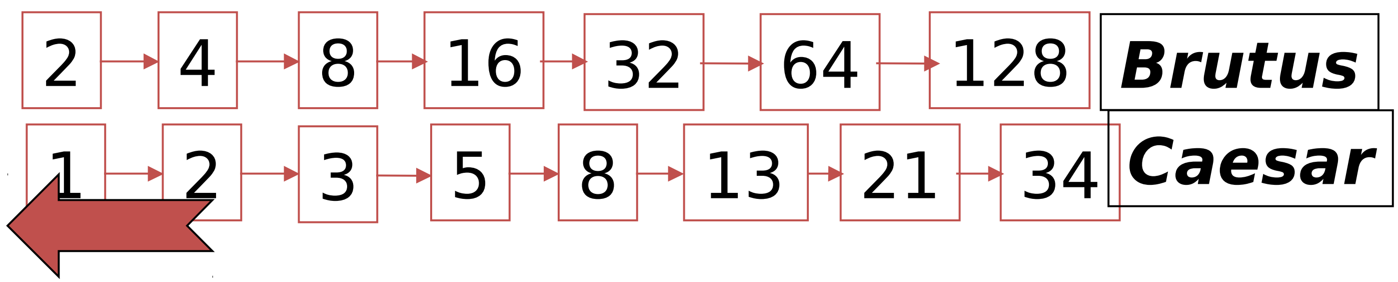
Brutus *AND* ***Caesar***

  - 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(p1, p2)
answer ← ⟨ ⟩
while p1 ≠ NIL and p2 ≠ NIL
do if docID(p1) = docID(p2)
      then ADD(answer, docID(p1))
           p1 ← next(p1)
           p2 ← next(p2)
      else if docID(p1) < docID(p2)
              then p1 ← next(p1)
              else p2 ← next(p2)
return answer
```

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.
  - 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

  <*term* : *docs*> entries

→ What can we do?

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
  - What is the **worst case for a biword index** in terms of dictionary size?
    For tri-gram (tri-word?)

- 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

- a) Extract inverted index entries for each distinct term: ***to, be, or, not.***
- b) 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

(part of) Homework

- **Implement a positional index**

- search in positional index.
  - Search with proximity query?

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*

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.

Positional index size

- Need an entry for each occurrence, not just once per document
- **Index size depends on average document size** Why?
  - 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 **(for web page)**
- Positional index size **35–50% of volume of original** text (for non-positional index maybe 10%)
  - 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 ¼ **of the time of using just a positional index by using some biwords**
  - It required **26% more space** than having a positional index alone
  - Modern systems – also add caching mechanisms