# Web Information Retrieval

Lecture 2

Tokenization, Normalization, Speedup, Phrase Queries

# Recap of the previous lecture

- Basic inverted indexes:
    - Structure: Dictionary and Postings
    - Key step in construction: Sorting
- Boolean query processing
    - Simple optimization
    - Linear time merging
- Overview of course topics

# Plan for this lecture

- Finish basic indexing
  - Tokenization
  - What terms do we put in the index?
- Query processing – speedups
- Proximity/phrase queries

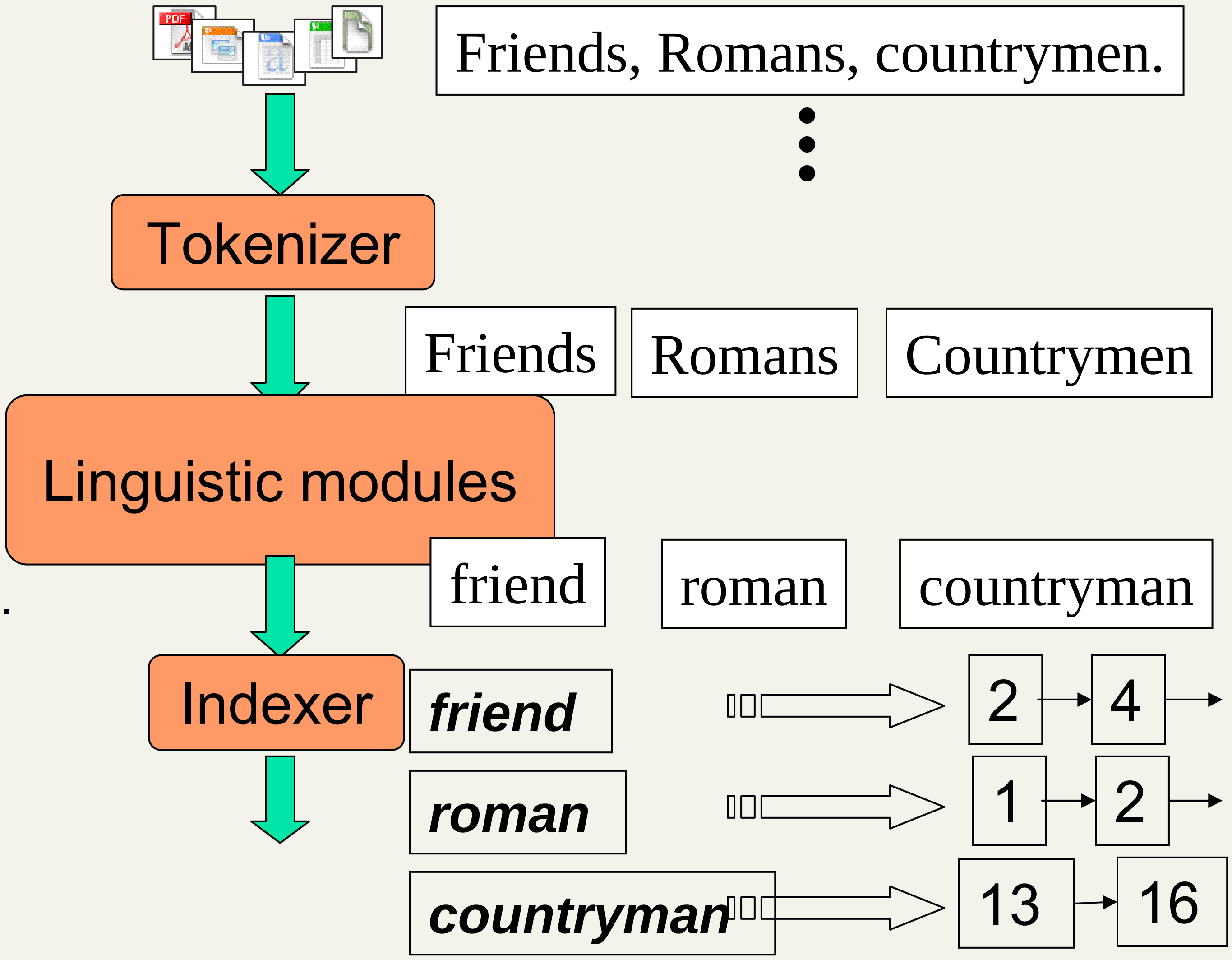
# Recall basic indexing pipeline

Documents to be indexed.

Token stream.

Modified tokens.

Inverted index.



Friends, Romans, countrymen.

Tokenizer

Friends Romans Countrymen

Linguistic modules

friend roman countryman

Indexer

*friend* → 2 → 4 →

*roman* → 1 → 2 →

*countryman* → 13 → 16

# Parsing a document

- What format is it in?
  - pdf/word/excel/html?
- What language is it in?
- What character set is in use?

Each of these is a classification problem.

But there are complications …

# Format/language stripping

- Documents being indexed can include docs from many different languages
  - A single index may have to contain terms of several languages.
- Sometimes a document or its components can contain multiple languages/formats
  - French email with a Portuguese pdf attachment.
- What is a unit document?
  - An email?
  - With attachments?
  - An email with a zip containing documents?

# Tokenization

# Tokenization

- Input: "***Friends, Romans and Countrymen***"
- Output: Tokens
  - ***Friends***
  - ***Romans***
  - ***Countrymen***
- Each such token is now a candidate for an index entry, after further processing
  - Described below
- But what are valid tokens to emit?

# Tokenization

- Issues in tokenization:
  - ***Finland's capital*** →

    ***Finland? Finlands? Finland's***?
  - ***Hewlett-Packard*** → ***Hewlett*** and ***Packard*** as two tokens?
    - ***State-of-the-art***: break up hyphenated sequence.
    - co-education ?
    - the hold-him-back-and-drag-him-away-maneuver ?
  - ***San Francisco***: one token or two?  How do you decide it is one token?

# Numbers

- ***3/12/91***
- ***Mar. 12, 1991***
- ***55 B.C.***
- ***B-52***
- ***My PGP key is 324a3df234cb23e***
- ***100.2.86.144***
  - Generally, don’t index as text.
  - Will often index “meta-data” separately
    - Creation date, format, etc.

# Tokenization: Language issues

- ***L'ensemble*** → one token or two?
  - ***L*** ? ***L'*** ? ***Le*** ?
  - Want ***ensemble*** to match with ***un ensemble***

- German noun compounds are not segmented
  - Lebensversicherungsgesellschaftsangestellter
  - 'life insurance company employee'

# Tokenization: language issues

- Arabic (or Hebrew) is basically written right to left, but with certain items like numbers written left to right
- Words are separated, but letter forms within a word form complex ligatures
- استقلت الجزائر في سنة 1962 بعد 132 عاما من الاحتلال الفرنسي.
- ‘Algeria achieved its independence in 1962 after 132 years of French occupation.’
- With Unicode, the surface presentation is complex, but the stored form is straightforward

# Normalization

- Need to “normalize” terms in indexed text as well as query terms into the same form
  - We want to match ***U.S.A.*** and ***USA***
- We most commonly implicitly define equivalence classes of terms
  - e.g., by deleting periods in a term

# Stop words

- With a stop list, you exclude from the dictionary entirely the commonest words. Intuition:
    - They have little semantic content: *the, a, and, to, be*
    - There are a lot of them: ~30% of postings for top 30 words
- But the trend is away from doing this:
    - Good compression techniques means the space for including stopwords in a system is very small
    - Good query optimization techniques mean you pay little at query time for including stop words.
    - You need them for:
        - Phrase queries: “King of Denmark”
        - Various song titles, etc.: “Let it be”, “To be or not to be”
        - “Relational” queries: “flights to London”

# Case folding

- Reduce all letters to lower case
  - exception: upper case (in mid-sentence?)
    - e.g., ***General Motors***
    - ***Fed*** vs. ***fed***
    - ***SAIL*** vs. ***sail***

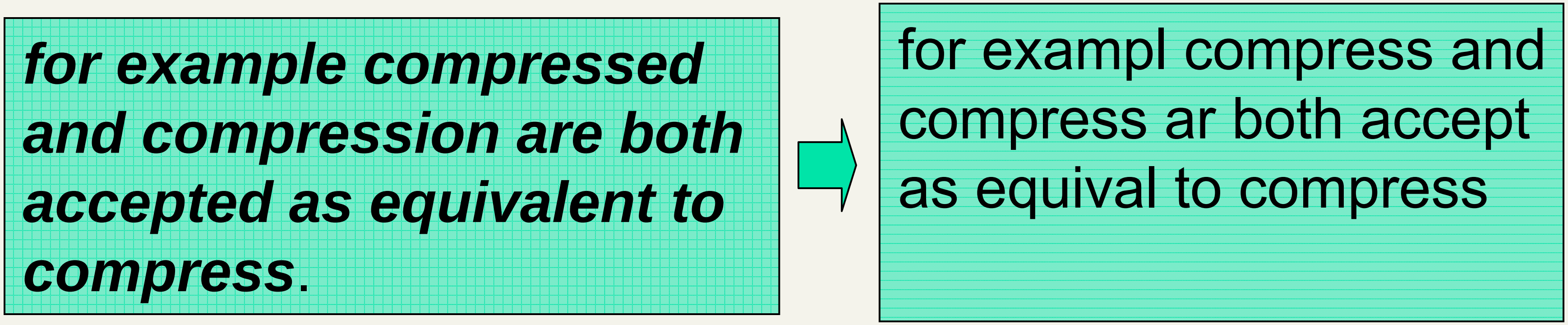
  - Often best to lower case everything, since users will use lowercase regardless of ‘correct’ capitalization

# Lemmatization

- Reduce inflectional/variant forms to base form
- E.g.,
  - *am, are, is* → *be*
  - *car, cars, car's, cars'* → *car*
- *the boy's cars are different colors* → *the boy car be different color*
- Lemmatization implies doing "proper" reduction to dictionary headword form

# Stemming

- Reduce terms to their “roots” before indexing
- “Stemming” suggest crude affix chopping
  - language dependent
  - e.g., ***automate(s), automatic, automation*** all reduced to ***automat***.



***for example compressed and compression are both accepted as equivalent to compress***.

for exampl compress and compress ar both accept as equival to compress

# Porter’s algorithm

- Commonest algorithm for stemming English
  - Results suggest at least as good as other stemming options
- Conventions + 5 phases of reductions
  - phases applied sequentially
  - each phase consists of a set of commands
  - sample convention: *Of the rules in a compound command, select the one that applies to the longest suffix.*

# Typical rules in Porter

- *sses → ss*
- *ies → i*
- *ational → ate*
- *tional → tion*

- Weight of word sensitive rules
- *(m>1) EMENT →*
  - *replacement → replac*
  - *cement → cement*

# Other stemmers

- Other stemmers exist, e.g., Lovins stemmer
  http://www.comp.lancs.ac.uk/computing/research/stemming/general/lovins.htm
  - Single-pass, longest suffix removal (about 250 rules)
  - Motivated by Linguistics as well as IR
- Full morphological analysis – at most modest benefits for retrieval
- Do stemming and other normalizations help?
  - Often very mixed results: really help recall for some queries but harm precision on others

# Language-specificity

- Many of the above features embody transformations that are
  - Language-specific and
  - Often, application-specific
- These are “plug-in” addenda to the indexing process
- Both open source and commercial plug-ins available for handling these

# Normalization: other languages

- Accents: ***résumé*** vs. ***resume***.
- Most important criterion:
  - How are your users like to write their queries for these words?
- Even in languages that standardly have accents, users often may not type them
- German: Tuebingen vs. Tübingen
  - Should be equivalent

# Normalization: other languages

- Need to “normalize” indexed text as well as query terms into the same form

***7-30 vs. 7/30***

- Character-level alphabet detection and conversion
  - Tokenization not separable from this.
  - Sometimes ambiguous:

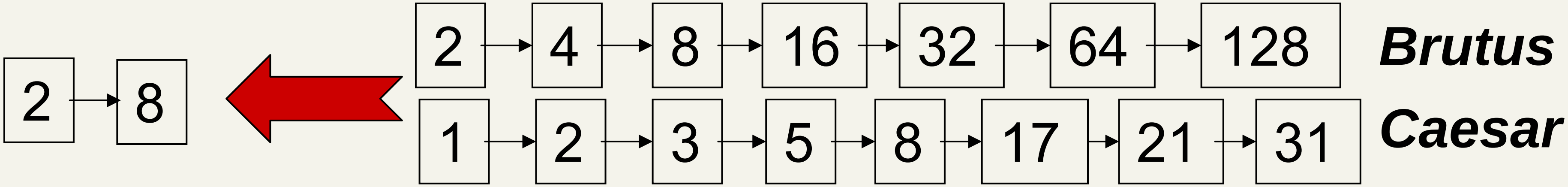
***Morgen will ich in MIT*** …

Is this German “mit”?

# Faster postings merges: Skip pointers

# Recall basic 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***

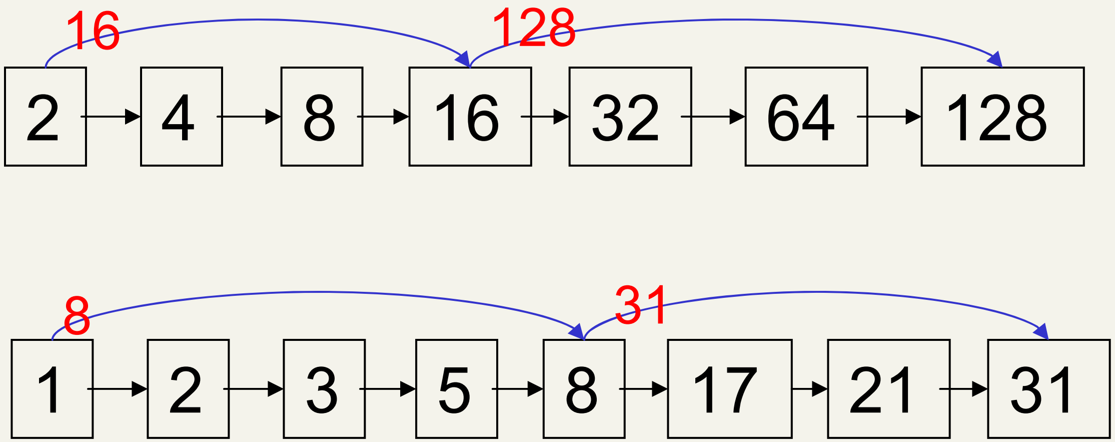
1 → 2 → 3 → 5 → 8 → 17 → 21 → 31 ***Caesar***

2 → 8

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

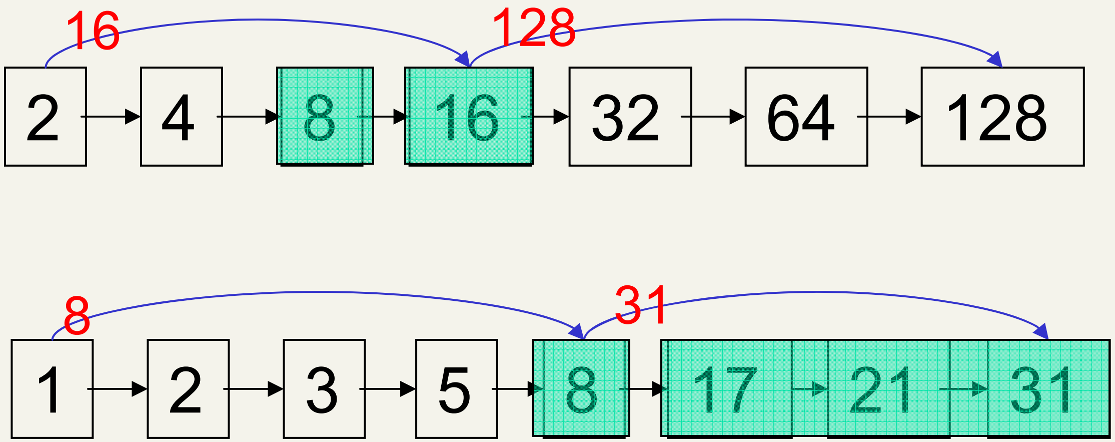
Can we do better?
Yes, if index isn't changing too fast.

# 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



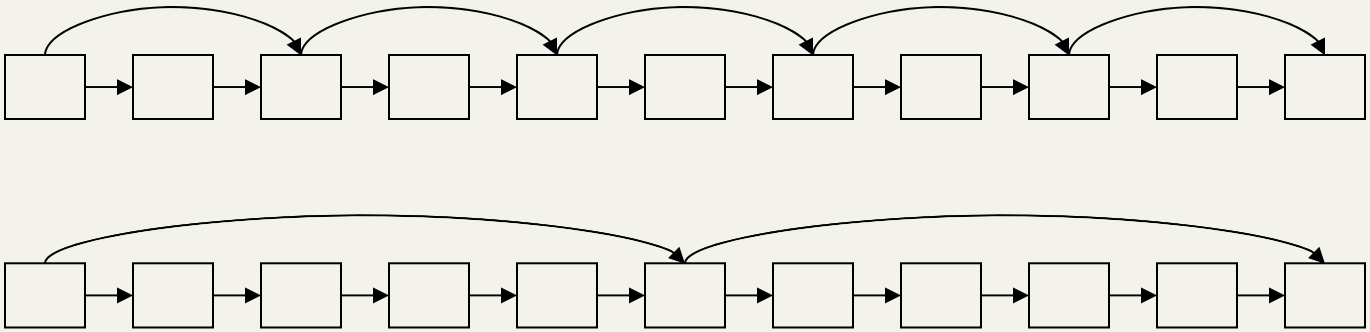
Suppose we've stepped through the lists until we process **8** on each list.

When we get to **16** on the top list, we see that its successor is **32**.

But the skip successor of **8** on the lower list is **31**, so we can skip ahead past the intervening postings.

# 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* 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)
  - The cost of loading a bigger postings list outweights the gain from quicker in memory merging

# Phrase queries

# Phrase queries

- Want to answer queries such as ***“villa adriana”*** – as a phrase
- Thus the sentence ***“adriana went to villa celimontana”*** is not a match.
  - The concept of phrase queries has proven easily understood by users; about 10% of web queries are phrase queries
- No longer suffices to store only

  <*term* : *docs*> entries

# 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 are processed as set of biwords:
- ***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

# Solution 2: Positional indexes

- Store, for each ***term***, entries of the form:

  <number of docs containing ***term***;

  *doc1*: position1, position2 … ;

  *doc2*: position1, position2 … ;

  etc.>

# Positional index example

<***be***: 993427;<br>
*1*: 7, 18, 33, 72, 86, 231;<br>
*2*: 3, 149;<br>
*4*: 17, 191, 291, 430, 434;<br>
*5*: 363, 367, …>

Which of docs 1,2,4,5 could contain "***to be or not to be***"?

- Can compress position values/offsets
- Nevertheless, this expands postings storage *substantially*

# 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***".

# Processing a phrase query

***to***, *993427*

*2*: 1,17,74,222,551;

*4*: 8,16,190,429,433;

*7*:13,23,191; ...

- ***be***, *178239*

  *1*: 17,19;

  *4*: 17,191,291,430,434;

  *5*: 14,19,101; ...

- Same general method for proximity searches

# Processing a phrase query

***to***, *993427*

*2*: 1,17,74,222,551;

*4*: 8,16,190,429,433;

*7*:13,23,191; ...

- ***be***, *178239*

  *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
  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*?

# Positional index size

- Can compress position values/offsets.
- Nevertheless, this expands postings storage *substantially*

# 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
- 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 ¼ of the time of using just a positional index
  - It required 26% more space than having a positional index alone

# Resources for today's lecture

- IIR Chapters 2.3, 2.4
- Porter's stemmer: http://www.tartarus.org/~martin/PorterStemmer/