Web Search:
Techniques, algorithms and Aplications

Basic Techniques for Web Search

German Rigau <german.rigau@ehu.es>

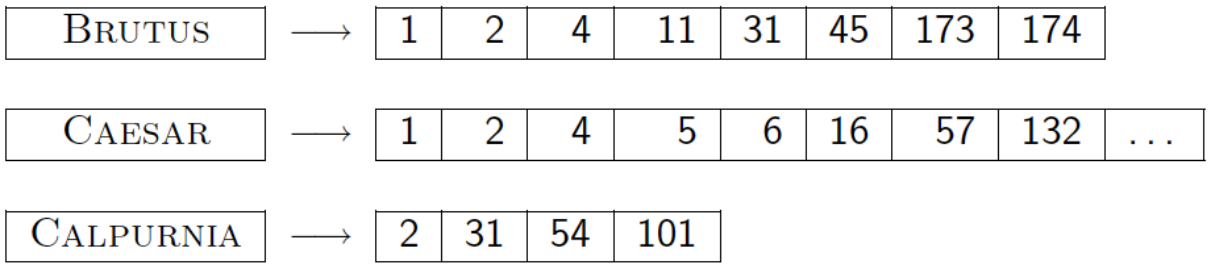
[Based on slides by Eneko Agirre …
and Christopher Manning and Prabhakar Raghavan]

Basic Techniques for Web Search

- Review of applications
- Basic Techniques in detail:
  - Boolean search
  - **Vocabularies, dictionaries**
  - Indexing
  - Scoring
  - complete system, evaluation
  - Web search
- Semantic search

Recap of the previous lectures

- Basic inverted indexes:
  - Structure: Dictionary and Postings

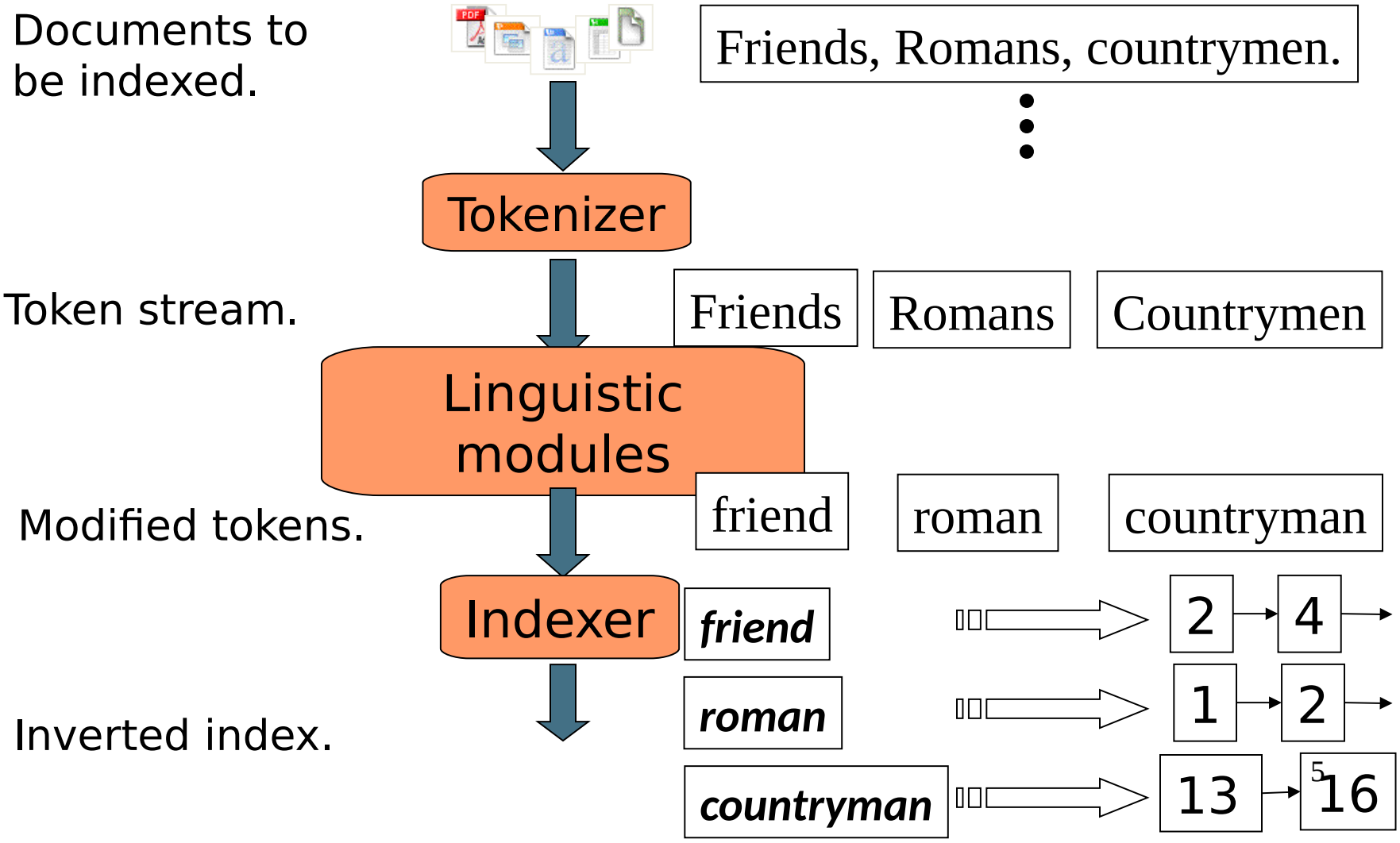


  - Key step in construction: **Sorting**
- Boolean query processing
  - Intersection by linear time "**merging**"
  - Simple optimizations

Terms and positions (ch. 2)

- Elaborate basic indexing
- Preprocessing to form the term vocabulary
  - Documents
  - Tokenization
  - What ***terms*** do we put in the index?
- Postings
  - Positional postings and phrase queries

Recall the basic indexing pipeline



Parsing a document

- What format is it in?
  - pdf/word/excel/html? … https://tika.apache.org
- What language is it in?
  - language identification
- What character set is in use?
  - UTF8, ...

- Each of these is a classification problem.
- But these tasks are often done heuristically …

Complications: Format/language

- 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 German pdf attachment.
- What is a unit document?
  - A file?
  - An email? (Perhaps one of many in an mbox.)
  - An email with 5 attachments?
  - A group of files (PPT or LaTeX as HTML pages)

Vocabularies, dictionaries

- **Tokens and Terms**
- Phrase Queries and Positional Indexes
- Dictionary data structures
- “Tolerant” retrieval

Tokenization

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

Token

- Input: "***Friends, Romans and Countrymen***"
- Output: Tokens
  - ***Friends***
  - ***Romans***
  - ***Countrymen***
- A token is an instance of a sequence of characters
- 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***
    - ***lowercase***, ***lower-case***, ***lower case*** ?
    - It can be effective to get the user to put in possible hyphens
  - ***San Francisco***: one token or two?
    - How do you decide it is one token?

Numbers

- ***3/20/91 Mar. 12, 1991 20/3/91***
- ***55 B.C.***
- ***B-52***
- ***My PGP key is 324a3df234cb23e***
- ***(800) 234-2333***
  - Often have embedded spaces
  - Older IR systems may not index numbers
    - But often very useful: think about things like looking up error codes/stacktraces on the web
    - (One answer is using n-grams)
  - Will often index “meta-data” separately
    - Creation date, format, etc.

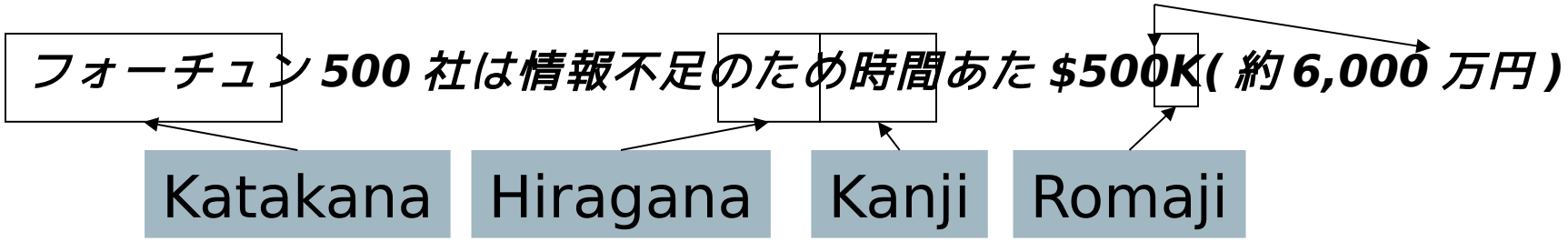
Tokenization: language issues

- French
  - ***L'ensemble*** → one token or two?
    - ***L*** ? ***L'*** ? ***Le*** ?
    - Want ***l'ensemble*** to match with ***un ensemble***
      - Until at least 2003, it didn't on Google
        - Internationalization!

- German noun compounds are not segmented
  - ***Lebensversicherungsgesellschaftsangestellter***
  - 'life insurance company employee'
  - German retrieval systems benefit greatly from a **compound splitter** module
    - Can give a 15% performance boost for German

Tokenization: language issues

- Chinese and Japanese have no spaces between words:
  - 莎拉波娃在居住在美国南部的佛里达。
  - Not always guaranteed a unique tokenization
- Further complicated in Japanese, with multiple alphabets intermingled
  - Dates/amounts in multiple formats



End-user can express query entirely in hiragana!

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.

- ← → ← → ← start
- '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

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”

Normalization to terms

- We need to “normalize” words in indexed text as well as query words into the same form
  - We want to match ***U.S.A.*** and ***USA***
- Result is terms: a term is a (normalized) word type, which is an entry in our IR system dictionary
- We most commonly implicitly define equivalence classes of terms by, e.g.,
  - deleting periods to form a term
    - ***U.S.A., USA => USA***
  - deleting hyphens to form a term
    - ***anti-discriminatory, antidiscriminatory => antidiscriminatory***

Normalization: other languages

- Accents: e.g., French ***résumé*** vs. ***resume*.**
- Umlauts: e.g., German: ***Tuebingen*** vs. ***Tübingen***
  - Should be equivalent
- 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
  - Often best to normalize to a de-accented term
    - ***Tuebingen, Tübingen, Tubingen => Tubingen***

Normalization: other languages

- Normalization of things like date forms
  - ***7 月 30 日 vs. 7/30***
  - ***Japanese use of kana vs. Chinese characters***

- Tokenization and normalization may depend on the language and so is intertwined with language detection

Morgen will ich in MIT …

Is this German “mit”?

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

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…
- Google example:
  - Query ***C.A.T.***
  - #1 result is for “cat” (well, Lolcats) *not* Caterpillar Inc.

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 it's 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)
- Full morphological analysis – at most modest benefits for retrieval
- Do stemming and other normalizations help?
  - English: very mixed results. Helps recall for some queries but harms precision on others
    - E.g., operative (dentistry) ⇒ oper
  - Definitely useful for Spanish, German, Finnish, Basque…
    - 30% performance gains for Finnish and Basque!

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 are available for handling these

Vocabularies, dictionaries

- Tokens and Terms
- **Phrase Queries and Positional Indexes**
- Dictionary data structures
- “Tolerant” retrieval

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

<*term* : *docs*> entries

A first attempt: Biword indexes

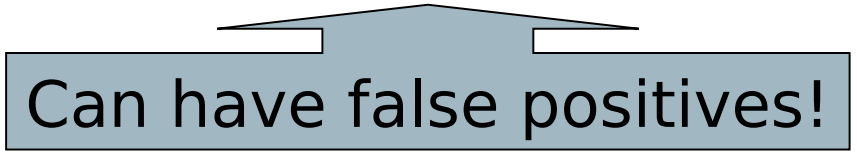
- Index every consecutive pair of terms in the text as a phrase
- For example the text “Friends, Romans, Countrymen” would generate the biwords
  - ***friends romans***
  - ***romans countrymen***
- Each of these biwords is now a dictionary term
- Two-word phrase query-processing is now immediate.

Longer phrase queries

- Longer phrases:
- ***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.



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

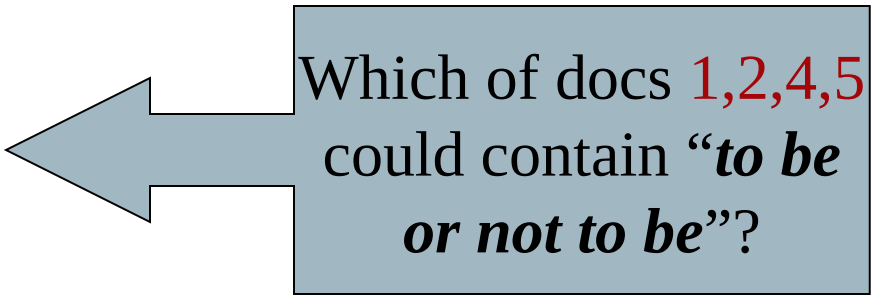
Solution 2: Positional indexes

- 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, …>



- 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

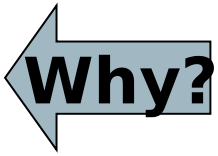
- LIMIT! /3 STATUTE /3 FEDERAL /2 TORT
  - Again, here, */k* means “within *k* words of”.
- Clearly, positional indexes can be used for such queries; biword indexes cannot.

Positional index size

- You can compress position values/offsets
- 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

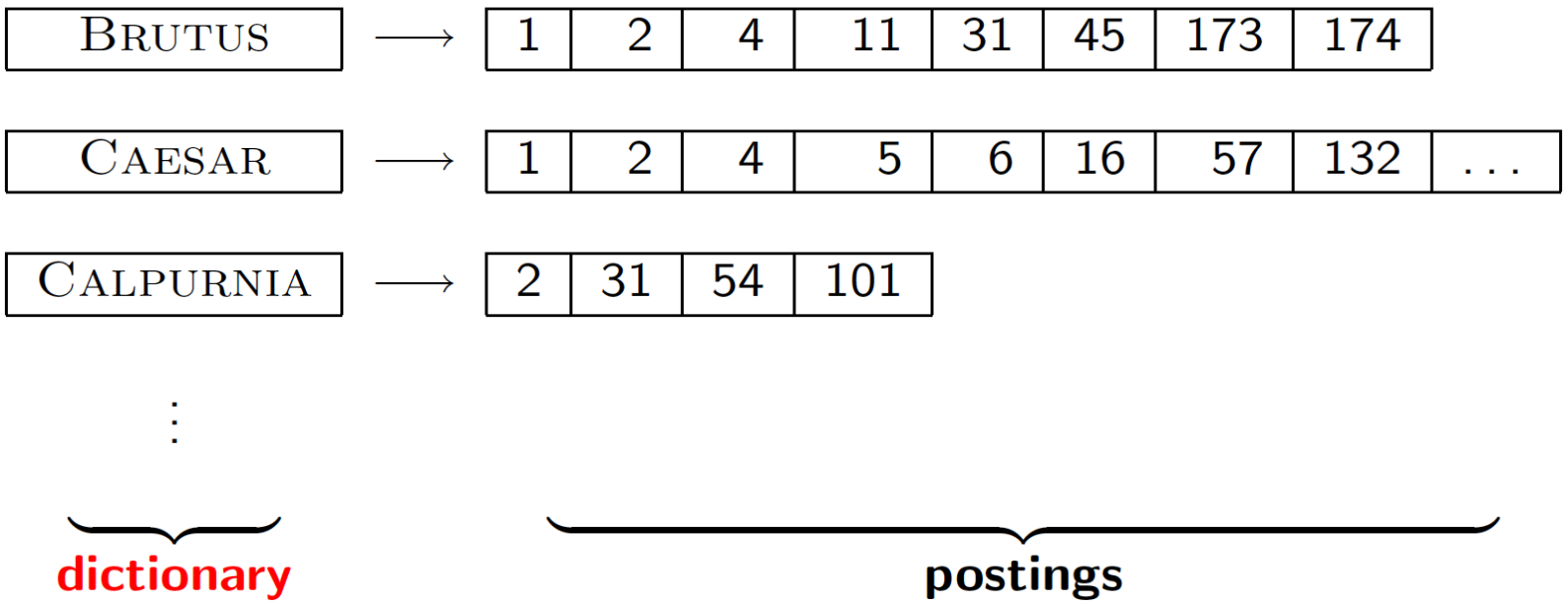
- A positional index is 2–4 as large as a non-positional index
- Positional index size 35–50% of volume of original text
- Caveat: all of this holds for “English-like” languages

Vocabularies, dictionaries

- Tokens and Terms
- Phrase Queries and Positional Indexes
- **Dictionary data structures**
- “Tolerant” retrieval

Dictionary data structures for inverted indexes

- The dictionary data structure stores the term vocabulary, document frequency, pointers to each postings list … in what data structure?



A naïve dictionary

- An array of struct:

term	document frequency	pointer to postings list
a	656,265	⟶
aachen	65	⟶
. . .	. . .	. . .
zulu	221	⟶

char[20] int Postings *

20 bytes 4/8 bytes 4/8 bytes

- How do we store a dictionary in memory efficiently?
- How do we quickly look up elements at query time?

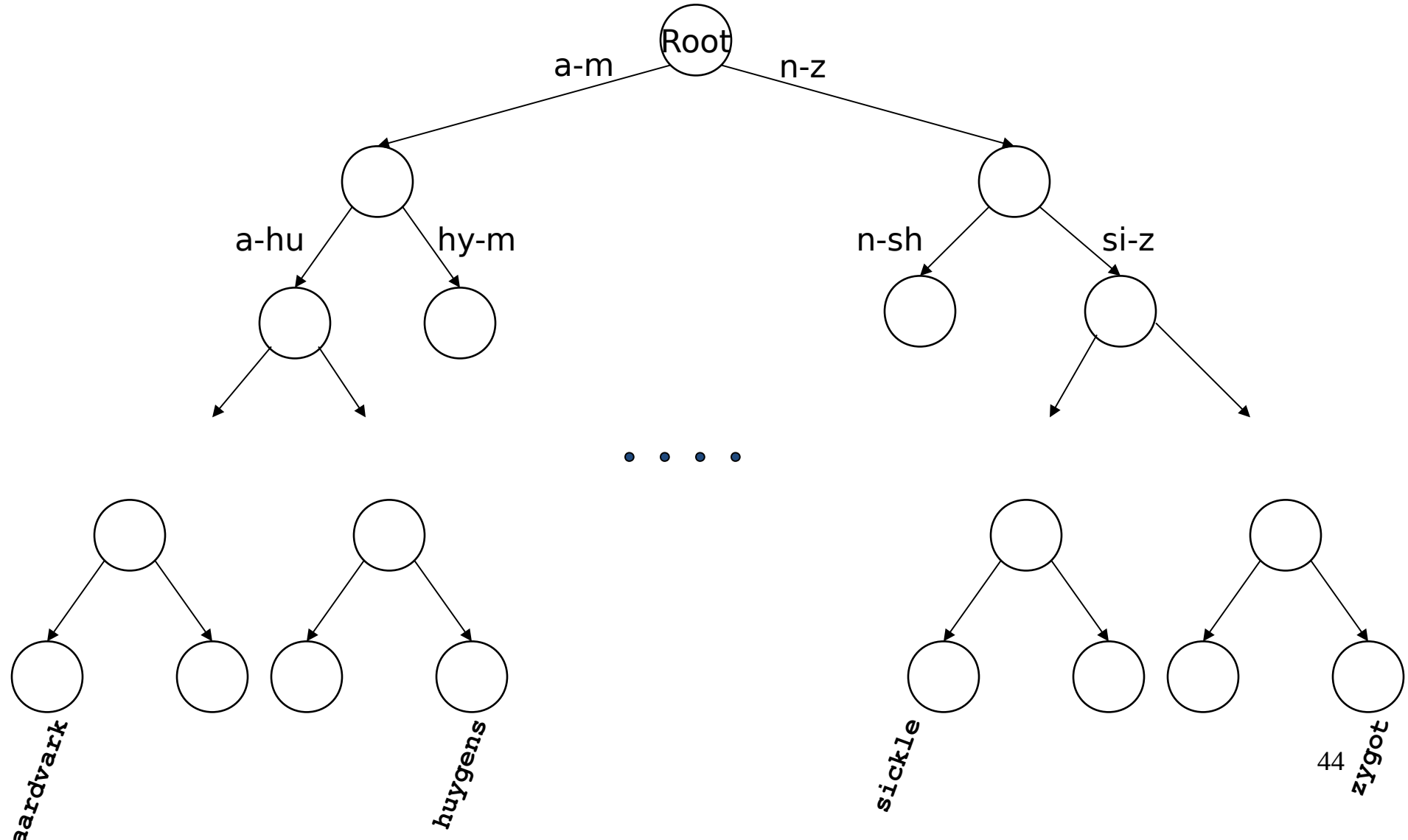
Dictionary data structures

- Two main choices:
  - Hash table
  - Tree
- Some IR systems use hashes, some trees

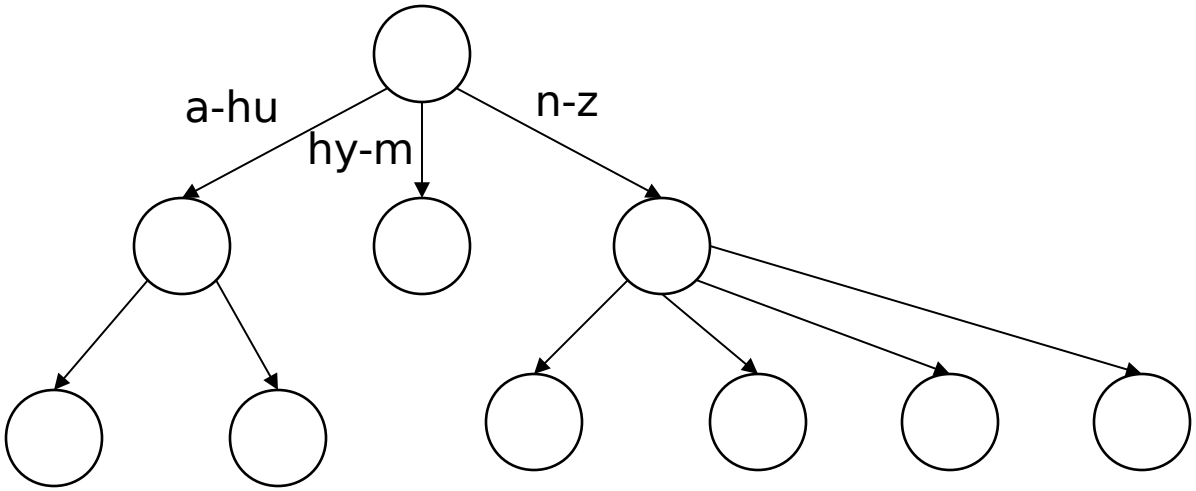
Hashes

- Each vocabulary term is hashed to an integer
  - (We assume you've seen hashtables before)
- Pros:
  - Lookup is faster than for a tree: O(1)
- Cons:
  - No easy way to find minor variants:
    - judgment/judgement
  - No prefix search [tolerant retrieval]
  - If vocabulary keeps growing, need to occasionally do the expensive operation of rehashing *everything*

Tree: binary tree



Tree: B-tree



- Definition: Every internal nodel has a number of children in the interval [*a*,*b*] where *a, b* are appropriate natural numbers, e.g., [2,4].

Trees

- Simplest: binary tree
- More usual: B-trees
- Trees require a standard ordering of characters and hence strings … but we standardly have one
- Pros:
  - Solves the prefix problem (terms starting with *hyp*)
- Cons:
  - Slower: O(log *M*) [and this requires *balanced* tree]
  - Rebalancing binary trees is expensive
    - But B-trees mitigate the rebalancing problem

Vocabularies, dictionaries

- Tokens and Terms
- Phrase Queries and Positional Indexes
- Dictionary data structures
- **“Tolerant” retrieval**
  - Wildcard queries
  - Spelling correction
  - Soundex

Vocabularies, dictionaries

- Tokens and Terms
- Phrase Queries and Positional Indexes
- Dictionary data structures
- “Tolerant” retrieval
  - **Wildcard queries**
  - Spelling correction
  - Soundex

Wild-card queries: *

- ***mon*:*** find all docs containing any word beginning “mon”.
- Easy with binary tree (or B-tree) lexicon: retrieve all words in range: ***mon ≤ w < moo***
- ****mon:*** find words ending in “mon”: harder
  - Maintain an additional B-tree for terms *backwards.*

Can retrieve all words in range: ***nom ≤ w < non***.

Exercise: from this, how can we enumerate all terms meeting the wild-card query ***pro*cent*** ?

Query processing

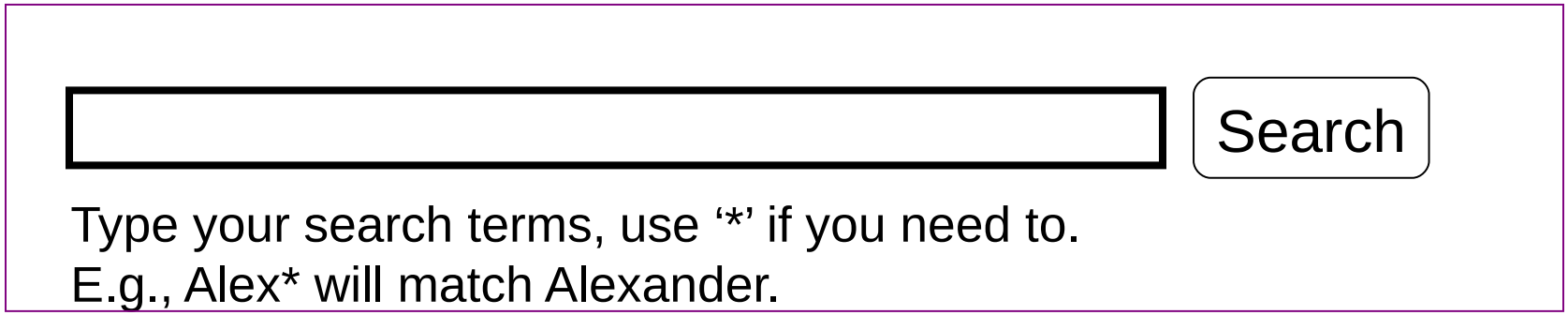
- At this point, we have an enumeration of all terms in the dictionary that match the wild-card query.
- We still have to look up the postings for each enumerated term.
- E.g., consider the query:

  se*ate *AND* ***fil*er***

  This may result in the execution of many Boolean *AND* queries.

Processing wild-card queries

- As before, we must execute a Boolean query for each enumerated, filtered term.
- Wild-cards can result in expensive query execution (very large disjunctions…)
  - pyth* AND prog*
- If you encourage “laziness” people will respond!



- Which web search engines allow wildcard queries?

Vocabularies, dictionaries

- Tokens and Terms
- Phrase Queries and Positional Indexes
- Dictionary data structures
- “Tolerant” retrieval
  - Wildcard queries
  - **Spelling correction**
  - Soundex

Spell correction

- Two principal uses
  - Correcting document(s) being indexed
  - Correcting user queries to retrieve “right” answers
- Two main flavors:
  - Isolated word
    - Check each word on its own for misspelling
    - Will not catch typos resulting in correctly spelled words
    - e.g., ***from → form***
  - Context-sensitive
    - Look at surrounding words,
    - e.g., ***I flew <u>form</u> Heathrow to Narita.***

Context-sensitive spell correction

- Text: ***I flew <u>from</u> Heathrow to Narita.***
- Consider the phrase query ***“flew <u>form</u> Heathrow”***
- We’d like to respond

  Did you mean “***flew from Heathrow***”?

because no docs matched the query phrase.

Context-sensitive correction

- Need surrounding context to catch this.
- First idea: retrieve dictionary terms close (in weighted edit distance) to each query term
- Now try all possible resulting phrases with one word “fixed” at a time
  - ***flew from heathrow***
  - ***fled form heathrow***
  - ***flea form heathrow***
- **Hit-based spelling correction:** Suggest the alternative that has lots of hits.

General issues in spell correction

- We enumerate multiple alternatives for “Did you mean?”
- Need to figure out which to present to the user
- Use heuristics
  - The alternative hitting most docs
  - Query log analysis + tweaking
    - For especially popular, topical queries
- Spell-correction is computationally expensive
  - Avoid running routinely on every query?
  - Run only on queries that matched few docs

Vocabularies, dictionaries

- Tokens and Terms
- Phrase Queries and Positional Indexes
- Dictionary data structures
- “Tolerant” retrieval
  - Wildcard queries
  - Spelling correction
  - **Soundex**

Soundex

- Class of heuristics to expand a query into phonetic equivalents
  - Language specific – mainly for names
  - E.g., ***chebyshev*** → ***tchebycheff***
- Invented for the U.S. census … in 1918

Soundex – typical algorithm

- Turn every token to be indexed into a 4-character reduced form
- Do the same with query terms
- Build and search an index on the reduced forms
  - (when the query calls for a soundex match)

- http://www.creativyst.com/Doc/Articles/SoundEx1/SoundEx1.htm#Top

Soundex – typical algorithm

1. Retain the first letter of the word.
2. Change all occurrences of the following letters to '0' (zero):
   'A', E', 'I', 'O', 'U', 'H', 'W', 'Y'.
3. Change letters to digits as follows:

- B, F, P, V → 1
- C, G, J, K, Q, S, X, Z → 2
- D,T → 3
- L → 4
- M, N → 5
- R → 6

Soundex continued

1. Remove all pairs of consecutive digits.
2. Remove all zeros from the resulting string.
3. Pad the resulting string with trailing zeros and return the first four positions, which will be of the form <uppercase letter> <digit> <digit> <digit>.

E.g., ***Herman*** becomes H655.

Will ***hermann*** generate the same code?

Soundex

- Soundex is the classic algorithm, provided by most databases (Oracle, Microsoft, …)
- How useful is soundex?
- Not very – for information retrieval
- Okay for “high recall” tasks (e.g., Interpol), though biased to names of certain nationalities
- Zobel and Dart (1996) show that other algorithms for phonetic matching perform much better in the context of IR

What queries can we process?

- We have
  - Positional inverted index
  - Wild-card index
  - Spell-correction
  - Soundex
- Queries such as

(SPELL(moriset) /3 toron*to) OR
*SOUNDEX****(chaikofski)***

Web Search:
Techniques, algorithms and Aplications

Basic Techniques for Web Search

German Rigau <german.rigau@ehu.es>

[Based on slides by Eneko Agirre …
and Christopher Manning and Prabhakar Raghavan]