Lecture 2: Encoding models for text

Many thanks to Prabhakar Raghavan for sharing most content from the following slides

Recap of the previous lecture

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

Plan for this lecture

Elaborate basic indexing

- Preprocessing to form the term vocabulary
  - Documents
  - Tokenization
  - What *terms* do we put in the index?
- Postings
  - Faster merges: skip lists
  - Positional postings and phrase queries
- Dictionary data structures

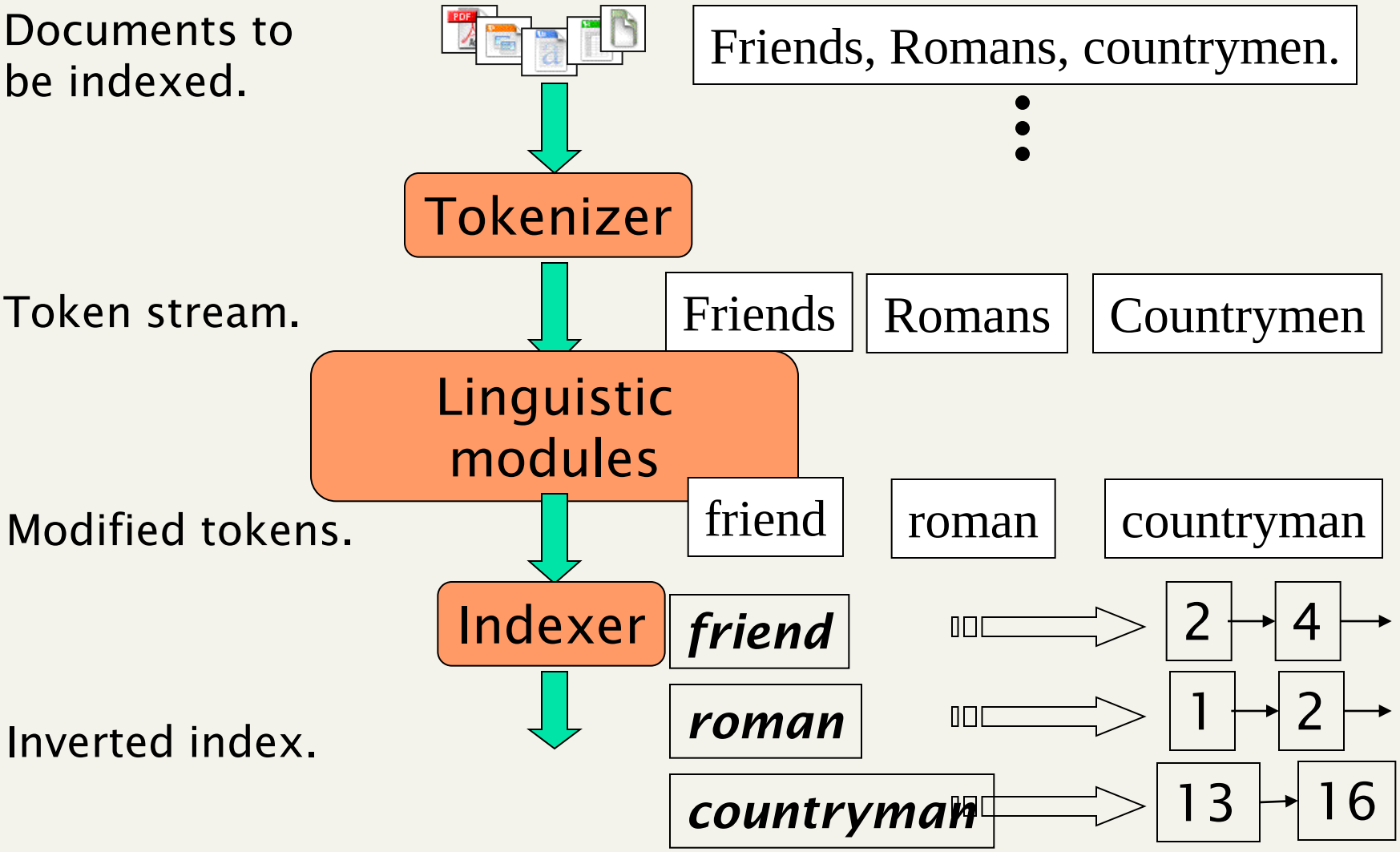
"Tolerant" retrieval

  - Wild-card queries
  - Spelling correction
  - Soundex

Basic indexing tools

- **Lucene:**
  - Provides core indexing and searching capabilities
    - Document formats are read using third-party libraries
  - Open source (i.e., you can adapt it to your project)
  - http://lucene.apache.org/
- **Lucy, Lucene.Net:**
  - Ports of Lucene to C and .Net
- **Nutch:**
  - *Web search engine* built on top of Lucene
  - Add crawler, link database, HTML parser, etc.
- **Solr:**
  - *Enterprise search engine* also building on a Lucene core
  - Adds faceted search, connector, administration features, etc.

Recall basic indexing pipeline



Parsing a document

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

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
  - Romanian email with an English 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)

Document Units

- Some issues to consider:
  - Multi-file formats (e.g., MBOX)
  - Books … too large for a single unit
  - Chinese/Japanese documents
  - etc.

Common Document Parsers

- HTML:
  - JTidy (fast, cool):
    - http://jtidy.sourceforge.net/
- PDF:
  - PDFBox (slow, but okay):
    - http://pdfbox.apache.org/
- Word (DOC), Excel (XLS):
  - Jakarta POI (not great, but best I know):
    - http://poi.apache.org/
- XML:
  - SAX Parser (popular, not too fast):
    - http://www.saxproject.org/about.html
    - http://java.sun.com/j2se/1.4.2/docs/api/javax/xml/parsers/SAXParser.html

Getting some test data

- TREC (Text Retrieval Conference)
  - (Kind of) free
  - http://trec.nist.gov/data.html
- Linguistic Data Consortium
  - Good data, not so cheap (0.5-5K$)
  - http://www.ldc.upenn.edu/Catalog/project_index.jsp
- Web search APIs
  - Yahoo!:
    - http://developer.yahoo.com/search/boss/ (best at this moment)
    - http://developer.yahoo.com/everything.html
  - Google:
    - http://code.google.com/intl/ro-RO/apis/customsearch/ (limited to your sites)
    - http://code.google.com/intl/ro-RO/apis/websearch/ (deprecated)

Language & Character set

- Use specialized dictionary to determine
- First 100 words of each document are usually sufficient to roughly isolate a clear language & character set
  - Use ~1,000 for high precision

Tokens and Terms

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***
    - ***lowercase***, ***lower-case***, ***lower case*** ?
    - It's 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

- *3/12/91* *Mar. 12, 1991*
- *55 B.C.*
- *B-52*
- *My PGP key is 324a3df234cb23e*
- *(800) 234-2333*
  - Often have embedded spaces
  - Often, don't index as text
    - But often very useful: think about things like looking up error codes / stack traces on the web
    - (One answer is using n-grams: Coming up later)
  - Will often index "meta-data" separately
    - Creation date, format, etc.

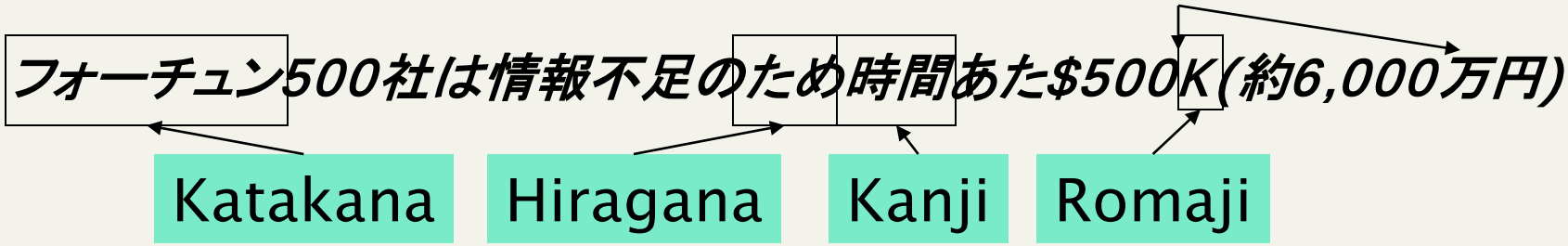
Tokenization: language issues

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

- 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
    - https://github.com/dweiss/compound-splitter

Tokenization: language issues

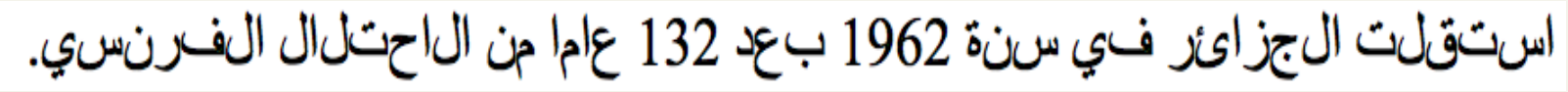
- Chinese and Japanese have no spaces between words:
  - 莎拉波娃现在居住在美国东南部的佛罗里达。
  - Not always guaranteed a unique tokenization
- Further complicated in Japanese, with multiple alphabets intermingled
  - Dates/amounts in multiple formats



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



- ← → ← → ← 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 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 wds
- But the trend is away from doing this:
  - Good compression techniques (next lecture) means the space for including stop words 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

- 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
- Alternative is to do asymmetric expansion:
  - Enter: ***window*** Search: ***window, windows***
  - Enter: ***windows*** Search: ***Windows, windows, window***
  - Enter: ***Windows*** Search: ***Windows***
- Potentially more powerful, but less efficient

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月30日 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”?

- Google example:
  - Query ***C.A.T.***
  - #1 result used to be for “cat” *not* Caterpillar Inc.

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…

Thesauri and soundex

- Handle synonyms and homonyms
  - Hand-constructed equivalence classes
    - e.g., *car* = *automobile*
    - *color* = *colour*
- Rewrite to form equivalence classes
- Index such equivalences
  - When the document contains ***automobile***, index it under ***car*** as well (usually, also vice-versa)
- Or expand query?
  - When the query contains ***automobile***, look under ***car*** as well

WordNet

- A lexical database for English
  - http://wordnet.princeton.edu/
  - Offers part-of-speech tags, synonyms, hierarchical relationships such as part-of, etc.
  - Open source, free
- EU is attempting now to create corresponding corpora for West European languages
  - http://www.illc.uva.nl/EuroWordNet/
  - Not free ☹

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
- Which is another (better) way to identify the best form of a query?

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)

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 $\rightarrow$ 1
- C, G, J, K, Q, S, X, Z $\rightarrow$ 2
- D,T $\rightarrow$ 3
- L $\rightarrow$ 4
- M, N $\rightarrow$ 5
- R $\rightarrow$ 6

Soundex continued

4. Remove all pairs of consecutive digits.
5. Remove all zeros from the resulting string.
6. 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

Soundex (resources)

- Good overview:
  - http://en.wikipedia.org/wiki/Soundex
- Implementation:
  - http://commons.apache.org/codec/apidocs/org/apache/commons/codec/language/Soundex.html

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*

Stemming Tools

- Snowball stemmer
  - Covers quite a lot of languages: English, French, Spanish, Portuguese, Italian, **ROMANIAN (!!!)**, German, Dutch, Swedish, Norwegian, Danish, Russian, Finnish, Hungarian, Turkish
  - Works well with Lucene
  - http://snowball.tartarus.org/

Other stemmers

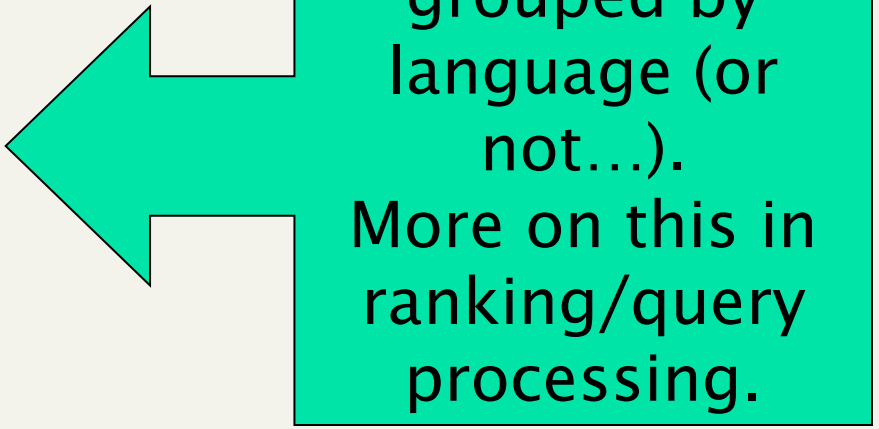
- Other stemmers exist, e.g., Lovins stemmer
  http://www.comp.lancs.ac.uk/computing/research/stemming/general/lovins.htm
  - Single-pass, longest suffix removal (about 250 rules)
- Full morphological analysis – at most modest benefits for retrieval
- Do stemming and other normalizations help?
  - English: very mixed results. Helps recall for some queries but harms precision on others
    - E.g., operative (dentistry) ⇒ oper
  - Definitely useful for Spanish, German, Finnish, …
    - 30% performance gains for Finnish!

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

Dictionary entries – first cut

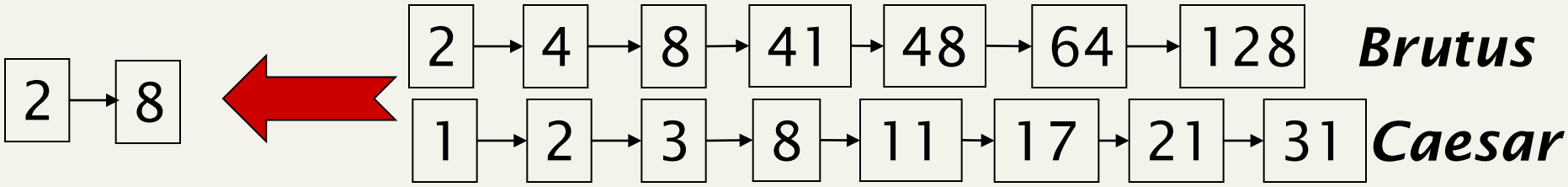
ensemble.french
時間.japanese
MIT.english
mit.german
guaranteed.english
entries.english
sometimes.english
tokenization.english



Faster postings merges: Skip pointers/Skip lists

Recall basic merge

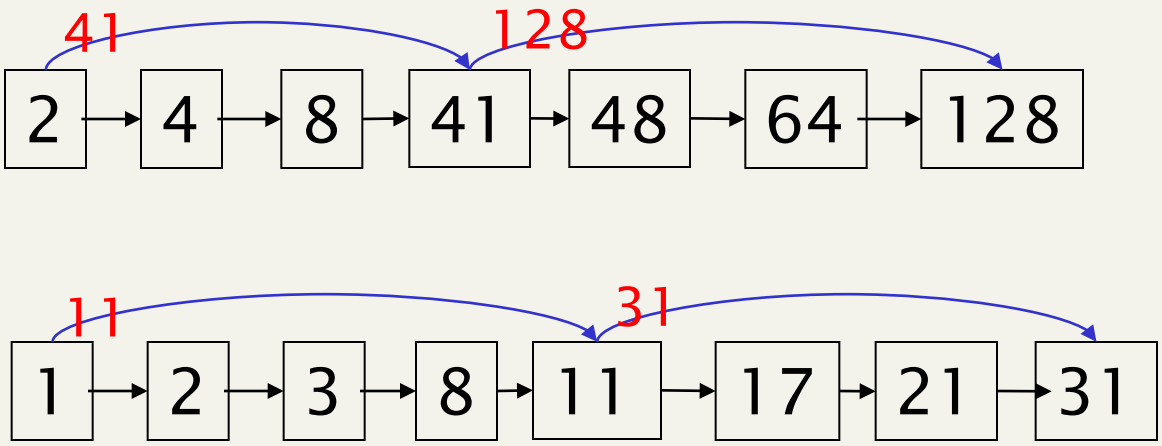
- Walk through the two postings simultaneously, in time linear in the total number of postings entries



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

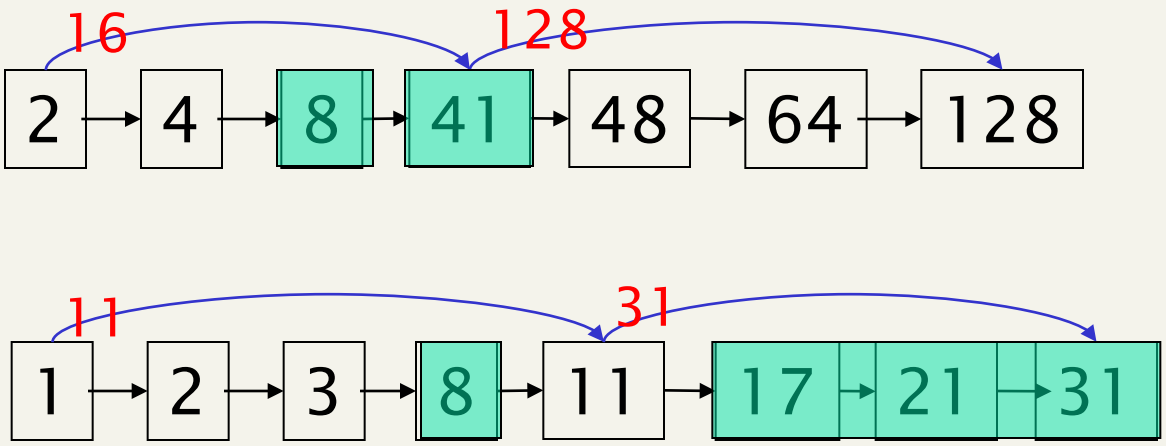
Can we do better?
Yes (if index isn't changing too fast).

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



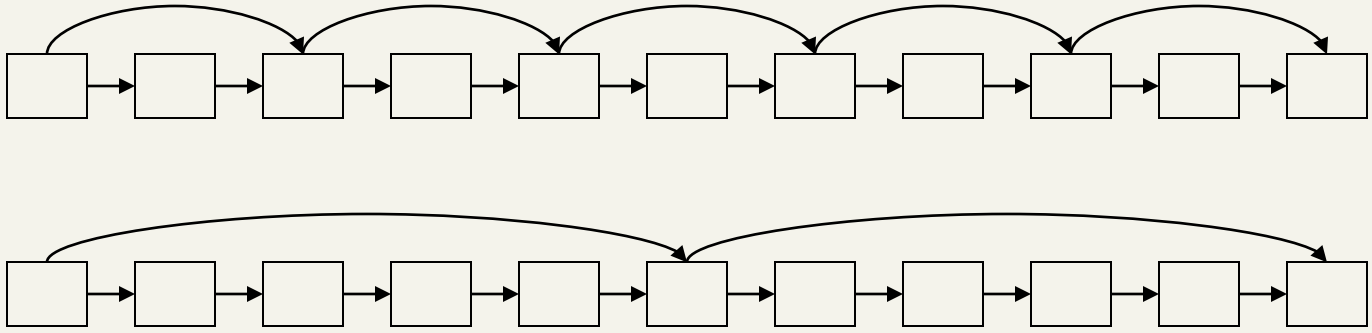
Suppose we've stepped through the lists until we process **8** on each list. We match it and advance.

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.

Where do we place skips?

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



Placing skips

- Simple heuristic: for postings of length L, use $\sqrt{L}$ 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 I/O cost of loading a bigger postings list can outweigh the gains from quicker in memory merging!

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

  <*term* : *docs*> entries

A first attempt: Biword indexes

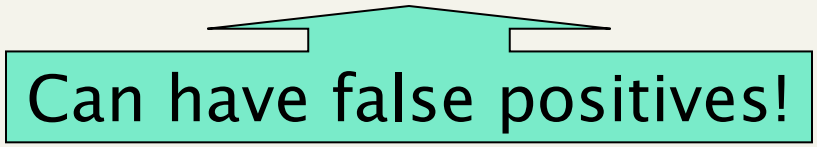
- Index every consecutive pair of terms in the text as a phrase
- For example the text “Friends, Romans, Countrymen” would generate the biwords
  - *friends romans*
  - *romans countrymen*
- Each of these biwords is now a dictionary term
- Two-word phrase query-processing is now immediate.

Longer phrase queries

- Longer phrases are processed as 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.



Extended biwords

- Parse the indexed text and perform part-of-speech-tagging (POST).
- Bucket the terms into (say) Nouns (N) and articles/prepositions (X).
- Now deem any string of terms of the form NX*N to be 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 index

Issues for biword indexes

- False positives, as noted before
- Index blowup due to bigger dictionary

- For extended biword index, parsing longer queries into conjunctions:
  - E.g., the query ***tangerine trees and marmalade skies*** is parsed into
  - ***tangerine trees*** *AND* ***trees and marmalade*** *AND* ***marmalade skies***

- Not standard solution (for all biwords)

Solution 2: Positional indexes

- In the postings, store, for each ***term***, entries of the form:

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

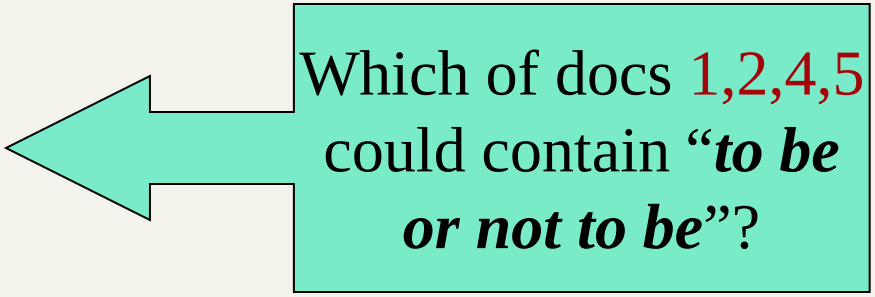
  doc1: position1, position2 … ;

  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, …>



- 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
  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

- You can compress position values/offsets: we'll talk about that later
- 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%

Why?

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

Rules of thumb

- A positional index is 2–4x 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

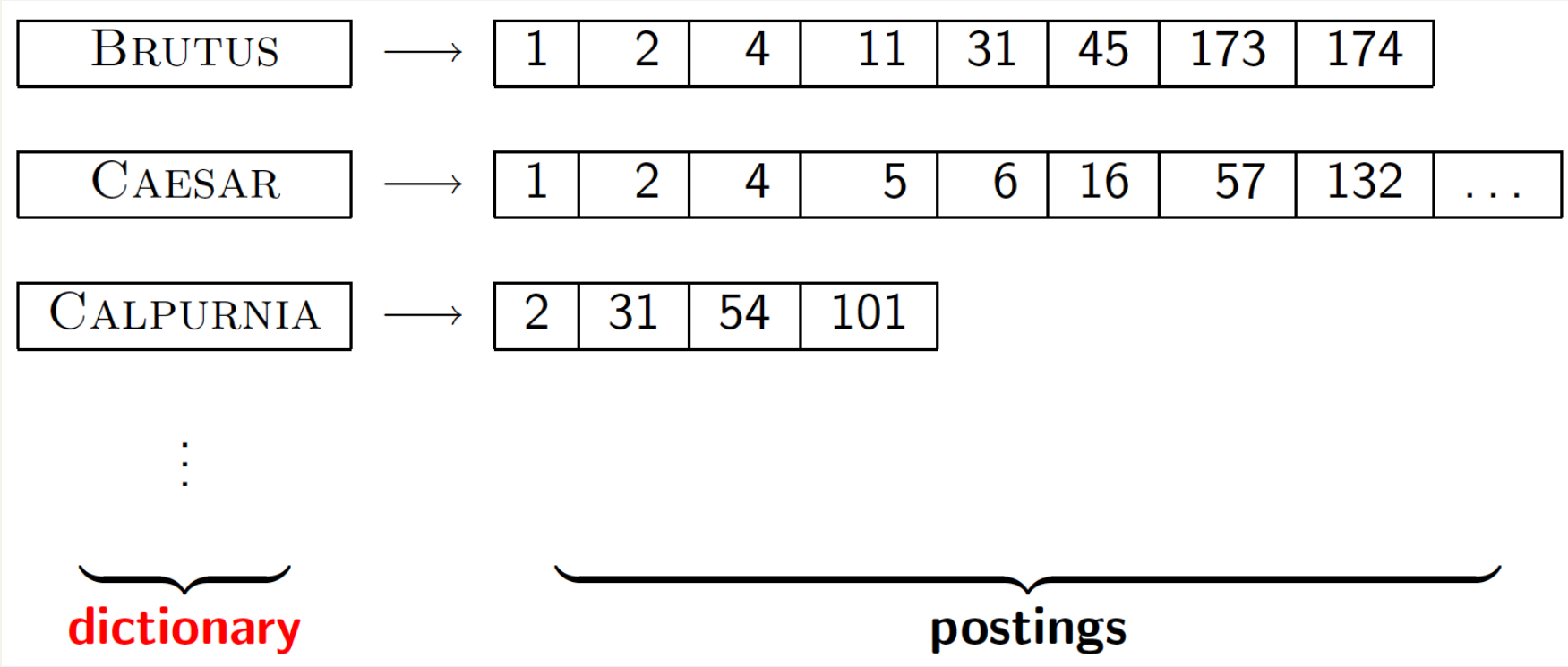
Combination schemes

- 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

Data Structures for Dictionaries

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?

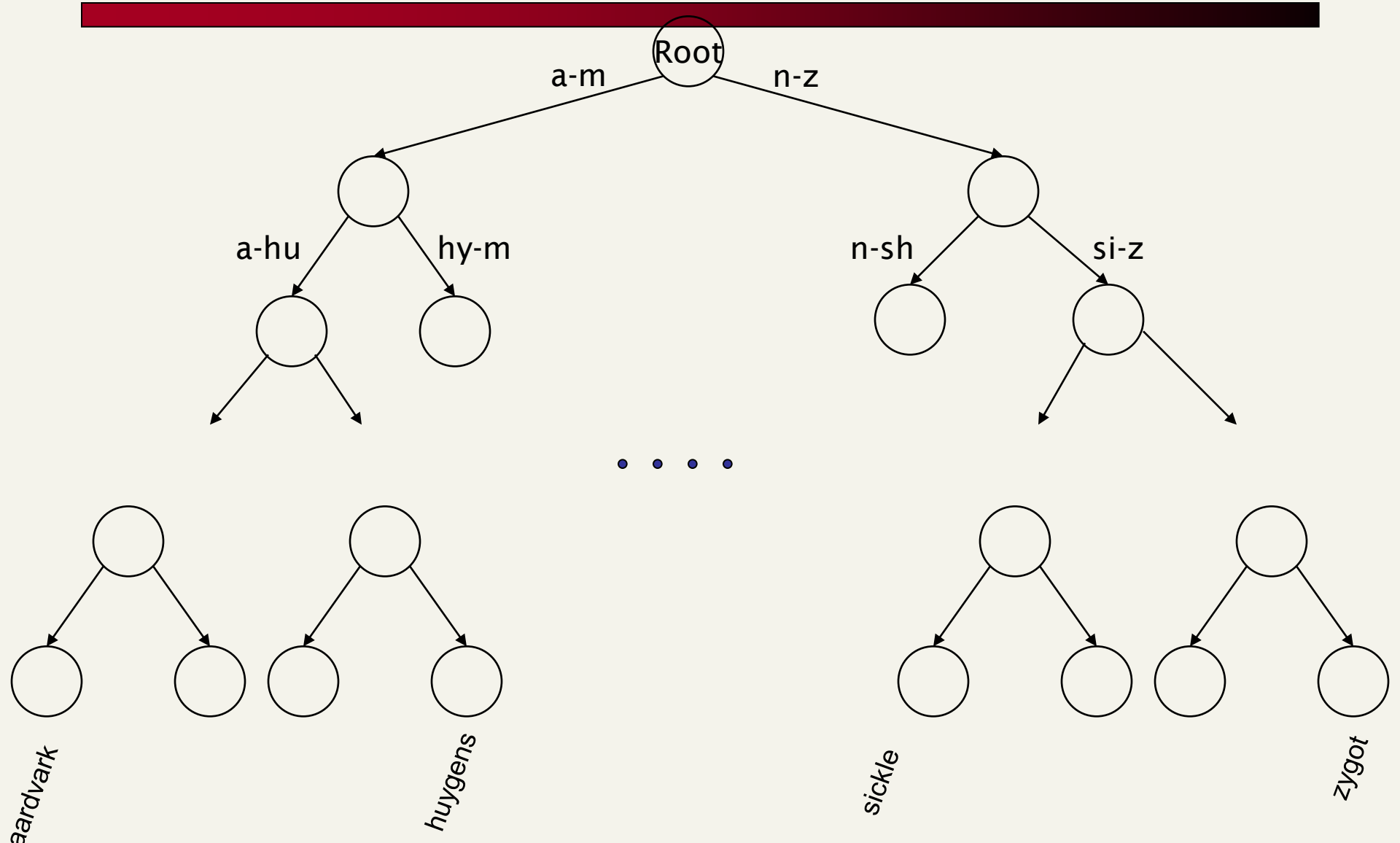
Dictionary data structures

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

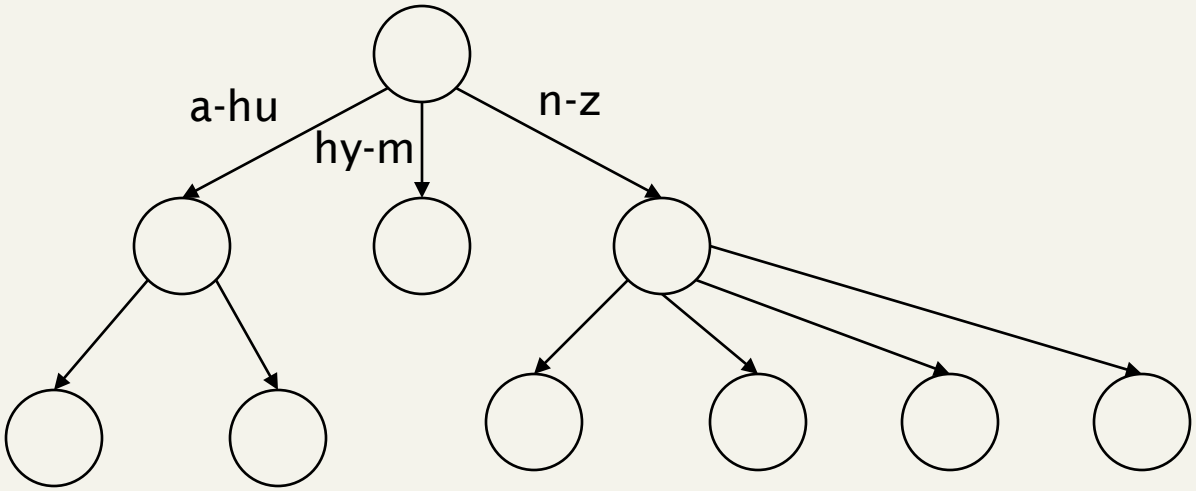
Hashes

- 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 going, 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

Wild-card queries

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.

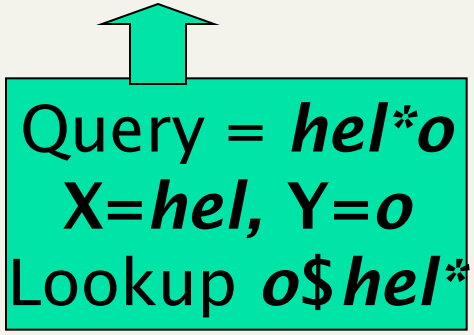
B-trees handle *'s at the end of a query term

- How can we handle *'s in the middle of query term?
  - *co*tion*
- We could look up *co** AND **tion* in a B-tree and intersect the two term sets
  - Expensive
- The solution: transform wild-card queries so that the *'s occur at the end
- This gives rise to the **Permuterm** Index.

Permuterm index

- For term ***hello***, index under:
  - ***hello$, ello$h, llohe, lohel, o$hell***

  where $ is a special symbol.
- Queries:
  - **X** lookup on **X$** **X*** lookup on **X*$**
  - ***X** lookup on **X$*** ***X*** lookup on **X***
  - **X*Y** lookup on **Y$X*** **X*Y*Z** ??? Exercise!



Permuterm example

- Query: m*n
- After the permuterm rotation is applied: n$m*
  - Hits: *man, men, moron*

Permuterm query processing

- Rotate query wild-card to the right
- Now use B-tree lookup as before.
- *Permuterm problem:* ≈ *quadruples lexicon size*

Empirical observation for English.

Bigram (*k*-gram) indexes

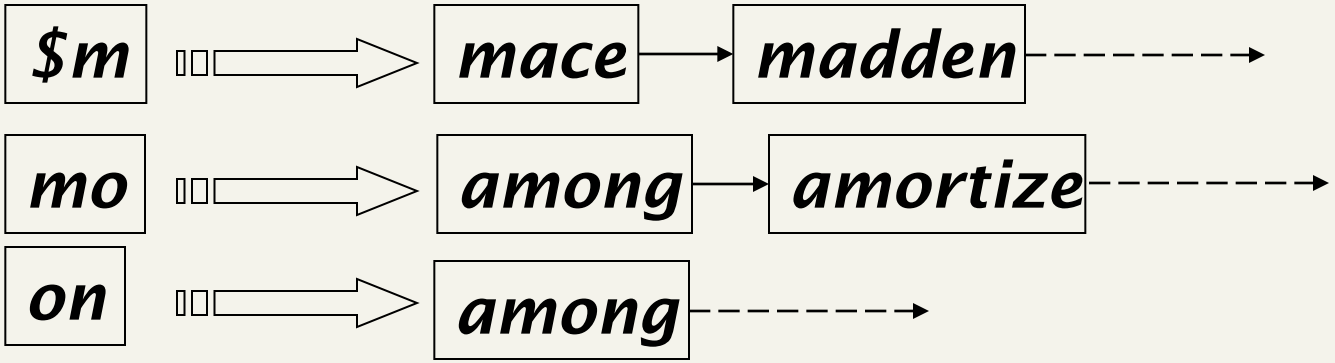
- Enumerate all *k*-grams (sequence of *k* chars) occurring in any term
- *e.g.,* from text "***April is the cruelest month***" we get the 2-grams (*bigrams*)

a,ap,pr,ri,il,l,i,is,s,t,th,he,e,$c,cr,ru,
ue,el,le,es,st,t$, m,mo,on,nt,h

  - $ is a special word boundary symbol
- Maintain a *second* inverted index *from bigrams to dictionary terms* that match each bigram.

Bigram index example

- The *k*-gram index finds *terms* based on a query consisting of *k-grams*

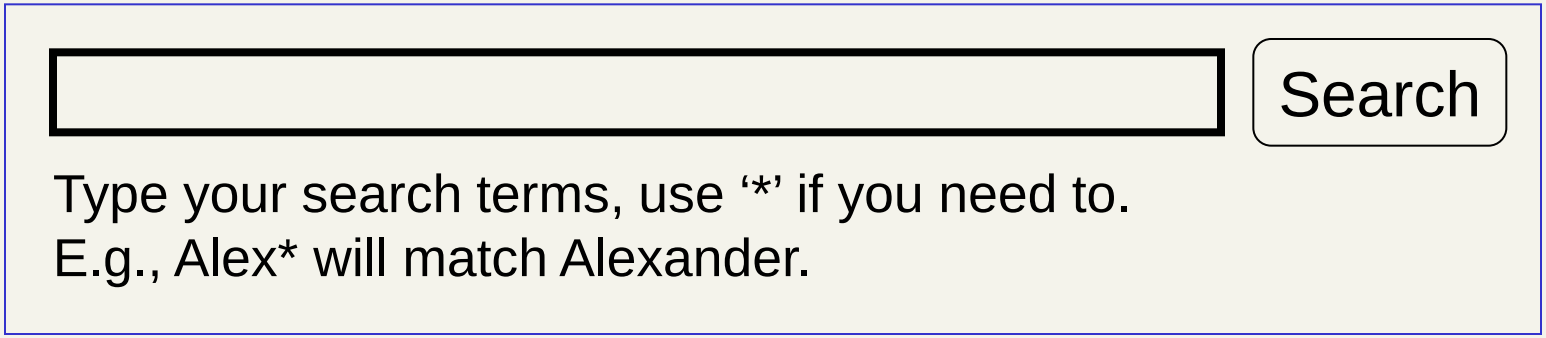


Processing *n*-gram wild-cards

- Query ***mon**** can now be run as
  - ***$m*** *AND* ***mo*** *AND* ***on***
- Gets terms that match AND version of our wildcard query.
- But we'd enumerate ***moon***.
- Must post-filter these terms against query.
- Surviving enumerated terms are then looked up in the term-document inverted index.
- Fast, space efficient (compared to permuterm).

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!



- Does Google allow wildcard queries?

Spelling correction

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 form Heathrow to Narita.*

Document correction

- Especially needed for OCR'ed documents
  - Correction algorithms are tuned for this: rn/m
  - Can use domain-specific knowledge
    - E.g., OCR can confuse O and D more often than it would confuse O and I (adjacent on the QWERTY keyboard, so more likely interchanged in typing).
- But also: web pages and even printed material has typos
- Goal: the dictionary contains fewer misspellings
- But often we don't change the documents but aim to fix the query-document mapping

Query mis-spellings

- Our principal focus here
  - E.g., the query ***Alanis Morisett***
- We can either
  - Retrieve documents indexed by the correct spelling, OR
  - Return several suggested alternative queries with the correct spelling
    - *Did you mean … ?*

Isolated word correction

- Fundamental premise – there is a lexicon from which the correct spellings come
- Two basic choices for this
  - A standard lexicon such as
    - Webster's English Dictionary
    - An "industry-specific" lexicon – hand-maintained
  - The lexicon of the indexed corpus
    - E.g., all words on the web
    - All names, acronyms etc.
    - (Including the mis-spellings)

Isolated word correction

- Given a lexicon and a character sequence Q, return the words in the lexicon closest to Q
- What's "closest"?
- We'll study several alternatives
  - Edit distance (Levenshtein distance)
  - Weighted edit distance
  - *n*-gram overlap

Edit distance

- Given two strings S_1 and S_2, the minimum number of operations to convert one to the other
- Operations are typically character-level
  - Insert, Delete, Replace, (Transposition)
- E.g., the edit distance from *dof* to *dog* is 1
  - From *cat* to *act* is 2 (Just 1 with transpose.)
  - from *cat* to *dog* is 3.
- Generally found by dynamic programming.
- See http://www.merriampark.com/ld.htm for a nice example plus an applet.

Weighted edit distance

- As above, but the weight of an operation depends on the character(s) involved
  - Meant to capture OCR or keyboard errors, e.g. *m* more likely to be mis-typed as *n* than as *q*
  - Therefore, replacing *m* by *n* is a smaller edit distance than by *q*
  - This may be formulated as a probability model
- Requires weight matrix as input
- Modify dynamic programming to handle weights

Using edit distances

- Given query, first enumerate all character sequences within a preset (weighted) edit distance (e.g., 2)
- Intersect this set with list of “correct” words
- Show terms you found to user as suggestions
- Alternatively,
  - We can look up all possible corrections in our inverted index and return all docs … slow
  - We can run with a single most likely correction
- The alternatives dis-empower the user, but save a round of interaction with the user

Edit distance to all dictionary terms?

- Given a (mis-spelled) query – do we compute its edit distance to every dictionary term?
  - Expensive and slow
  - Alternative?
- How do we cut the set of candidate dictionary terms?
- One possibility is to use *n*-gram overlap for this
- This can also be used by itself for spelling correction.

n-gram overlap

- Enumerate all the *n*-grams in the query string as well as in the lexicon
- Use the *n*-gram index (recall wild-card search) to retrieve all lexicon terms matching any of the query *n*-grams
- Threshold by number of matching *n*-grams
  - Variants – weight by keyboard layout, etc.

Example with trigrams

- Suppose the text is ***november***
  - Trigrams are *nov, ove, vem, emb, mbe, ber*.
- The query is ***december***
  - Trigrams are *dec, ece, cem, emb, mbe, ber*.
- So 3 trigrams overlap (of 6 in each term)
- How can we turn this into a normalized measure of overlap?

One option – Jaccard coefficient

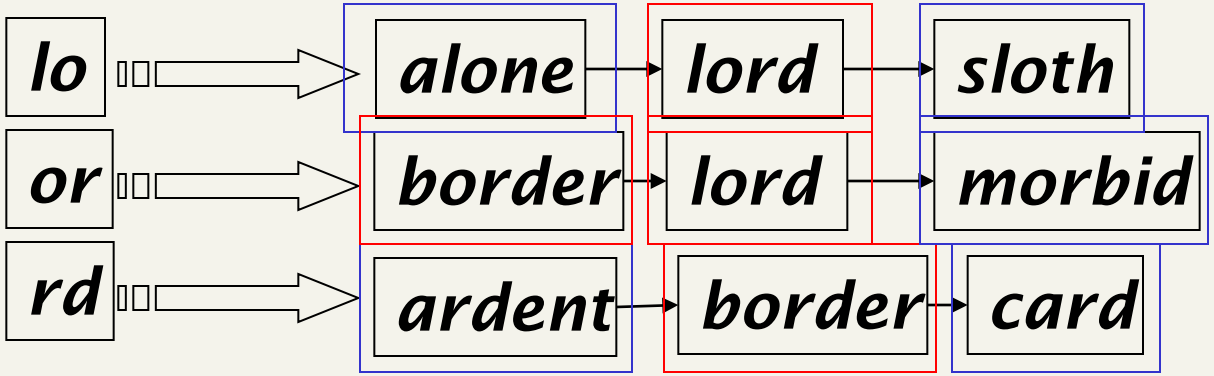
- A commonly-used measure of overlap
- Let X and Y be two sets; then the J.C. is

$$\left|X \cap Y\right| / \left|X \cup Y\right|$$

- Equals 1 when X and Y have the same elements and zero when they are disjoint
- X and Y don't have to be of the same size
- Always assigns a number between 0 and 1
  - Now threshold to decide if you have a match
  - E.g., if J.C. > 0.8, declare a match

Matching trigrams

- Consider the query ***lord*** – we wish to identify words matching 2 of its 3 bigrams (***lo, or, rd***)



Standard postings "merge" will enumerate …

Adapt this to using Jaccard (or another) measure.

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.

Exercise

- Suppose that for ***"flew form Heathrow"*** we have 7 alternatives for flew, 19 for form and 3 for heathrow.

How many "corrected" phrases will we enumerate in this scheme?

Another approach

- Break phrase query into a conjunction of biwords.
- Look for biwords that need only one term corrected.
- Enumerate phrase matches and … rank them!

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 (time based!), topical queries
- Spell-correction is computationally expensive
  - Avoid running routinely on every query?
  - Run only on queries that matched few docs
  - Cache

What queries can we process?

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

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

Exercise

- Draw yourself a diagram showing the various indexes in a search engine incorporating all the functionality we have talked about
- Identify some of the key design choices in the index pipeline:
  - Does stemming happen before the Soundex index?
  - What about *n*-grams?
- Given a query, how would you parse and dispatch sub-queries to the various indexes?

Resources for today's lecture

- WordNet DB for English words and semantics:
  - http://wordnet.princeton.edu/
- Skip Lists theory: Pugh (1990)
  - Multilevel skip lists give same O(log *n*) efficiency as trees

Resources for today's lecture

- Porter's stemmer: http://www.tartarus.org/~martin/PorterStemmer/
- Snowball stemming library: http://snowball.tartarus.org/
- Nice, easy reading on spell correction:
  - **Peter Norvig: How to write a spelling corrector** http://norvig.com/spell-correct.html

Even More Resources (trends in Information Retrieval)

- http://static.googleusercontent.com/external_content/untrusted_dlcp/research.google.com/en//people/jeff/WSDM09-keynote.pdf
- http://staff.science.uva.nl/~nicu/PUBS/PDF/2006/ACM-TOMCCAP-Feb06.pdf
- http://www.asiaa.sinica.edu.tw/~ccchiang/GILIS/LIS/p47-belkin.pdf
- [Off-topic] Introduction to Visual Sciences: http://videolectures.net/nips09_torralba_uvs/#

Other Resources (People Search)

- http://portal.acm.org/citation.cfm?id=1119181
- http://portal.acm.org/citation.cfm?id=1008992.1009040&coll=GUIDE&dl=ACM&type=series&idx=1008992&part=Proceedings&WantType=Proceedings&title=Annual%20ACM%20Conference%20on%20Research%20and%20Developmen
- http://portal.acm.org/citation.cfm?id=1621474.1621532
- http://portal.acm.org/citation.cfm?id=1621474.1621493
- http://portal.acm.org/citation.cfm?id=1060745.1060813
- http://ieeexplore.ieee.org/xpls/abs_all.jsp?arnumber=4221777&tag=1