CS3245

Information Retrieval

4

Lecture 4: Dictionaries and Tolerant Retrieval

Live Q&A
https://pollev.com/jin

Last Time: Postings lists and Choosing terms

- Faster merging of posting lists
  - Skip pointers
- Handling of phrase and proximity queries
  - Biword indexes for phrase queries
  - Positional indexes for phrase/proximity queries
- Steps in choosing terms for the dictionary
  - Text extraction
  - Granularity of indexing
  - Tokenization
  - Stop word removal
  - Normalization
  - Lemmatization and stemming

Today: the dictionary and tolerant retrieval

- Dictionary

- "Tolerant" retrieval
  - Wild-card queries
  - Spelling correction
  - Soundex

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?

BRUTUS	7	→	1	2	4	11	31	45	173	174
CAESAR	12	→	1	2	4	5	6	16	57	132	. . .
CALPURNIA	4	→	2	31	54	101				
⋮										

dictionary **postings**

A naïve dictionary

- An (possibly **unsorted**) array of entries:

	term	document frequency	pointer to postings list
dict[0]	a	656,265	⟶
dict[1]	aachen	65	⟶
…	. . .	. . .	. . .
dict[…]	zulu	221	⟶

`char[20]`	`int`	`Postings Pointer`
`20 bytes`	`8 bytes`	`8 bytes`

Quick Q: What's wrong with using this data structure?

A naïve dictionary

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

char[20]	int	Postings Pointer
20 bytes	8 bytes	8 bytes

- Words can only be at most 20 chars long. Waste of space for some words, not enough for others.
- How do we store a dictionary efficiently?
  → Later in W6

A naïve dictionary

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

char[20]	int	Postings Pointer
20 bytes	8 bytes	8 bytes

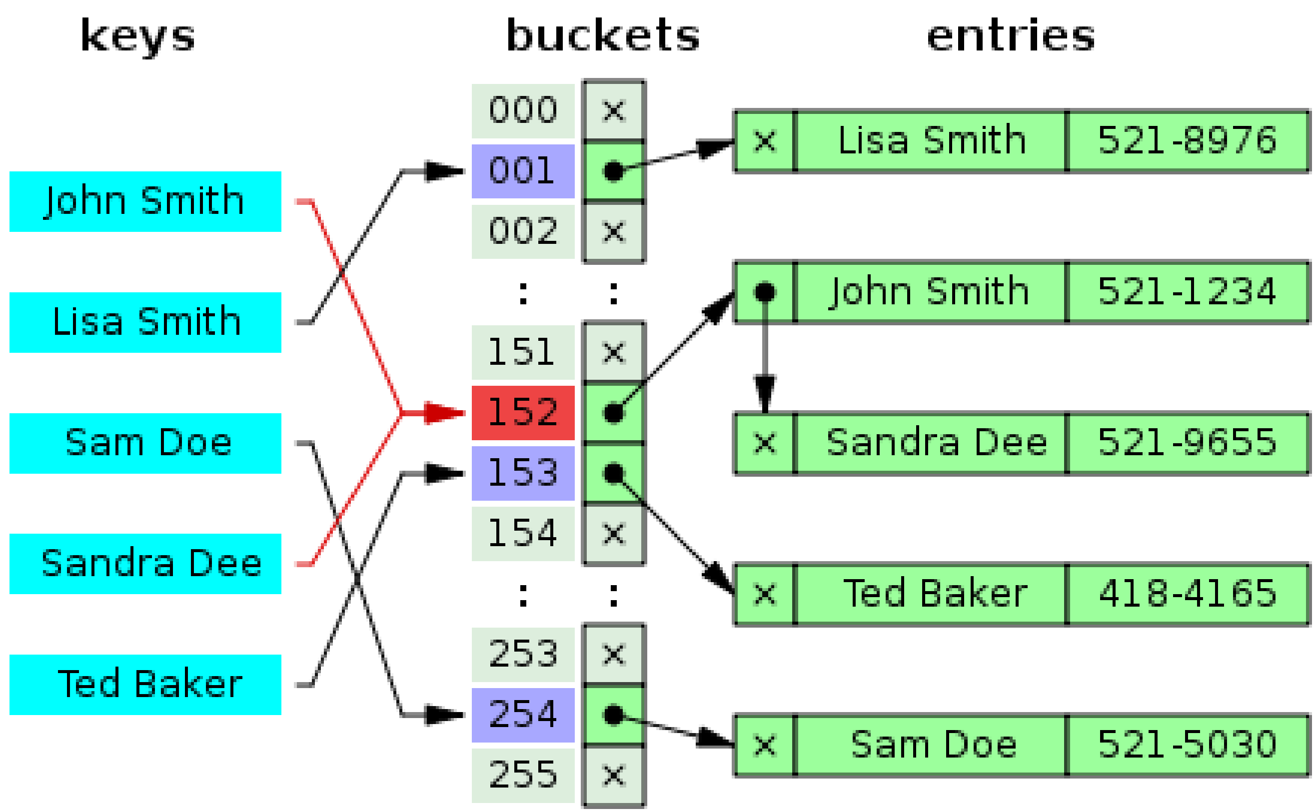
- Slow to access, linear scan needed!
- How do we quickly look up elements at query time?

Dictionary data structures

- Two main choices:
  - Hash table
  - Tree

- Focus on the support of tolerant retrieval for this lecture
  - See the textbook for other considerations!

Hash Table

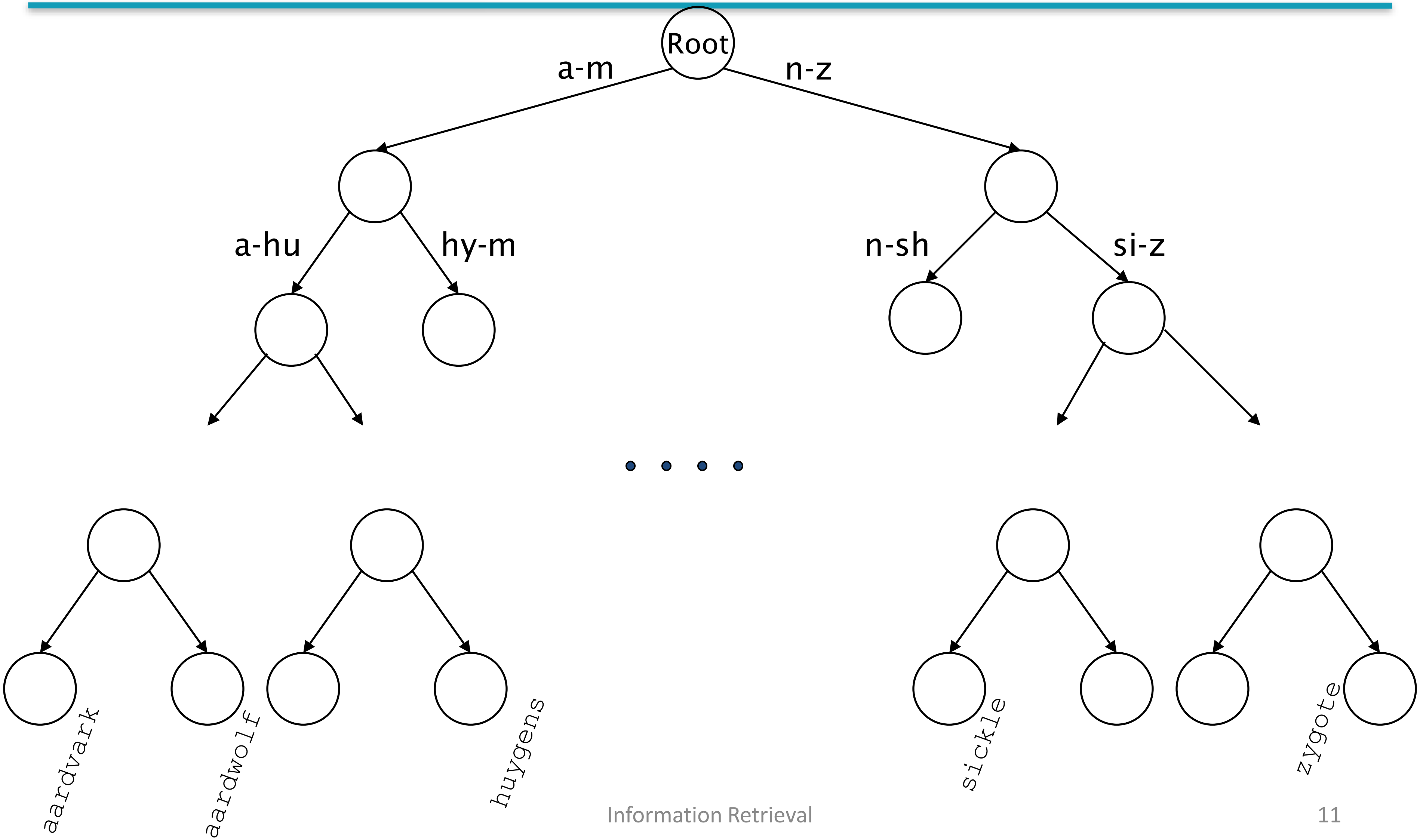

keys
buckets
entries
John Smith
Lisa Smith
Sam Doe
Sandra Dee
Ted Baker
000
001
002
151
152
153
154
253
254
255
Lisa Smith
521-8976
John Smith
521-1234
Sandra Dee
521-9655
Ted Baker
418-4165
Sam Doe
521-5030

Hash Table

- Pros:
  - Faster (than Tree): O(1) for lookup
- Cons:
  - No easy way to find minor variants:
    - judgment/judgement
  - No prefix search (e.g., terms starting with *"hyp"*)

Not very tolerant!

Tree: binary tree



Trees

- Pros:
  - Solves the prefix problem (e.g., terms starting with *"hyp"*)
  - Easier to find minor variants:
    - judgment/judgement

More tolerant!

- Cons:
  - Slower: O(log *M*) [and this requires a *balanced* tree]

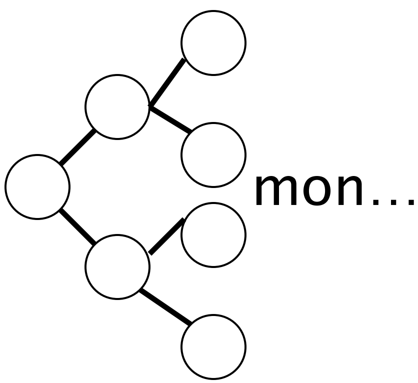
WILDCARD QUERIES

Wildcard queries: *

- * matches with any sequence of letters
- Sample use cases
  - File search based on extension (e.g., *.jpg)
  - Variation in spelling (e.g., col*ur)
  - Single vs plural form (e.g., cat*)
  - …

Wildcard queries: *

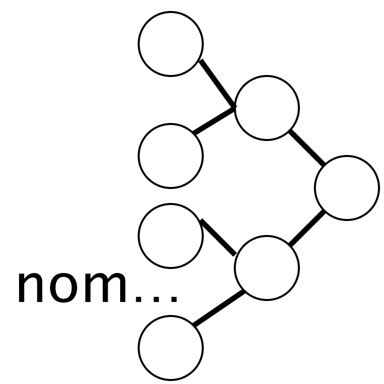
- ***mon*:*** find docs with words beginning with "mon".
  - Maintain a binary tree for terms
  - Retrieve all words in range: ***mon ≤ w < moo***



…
money
monsoon
month
…

Wildcard queries: *

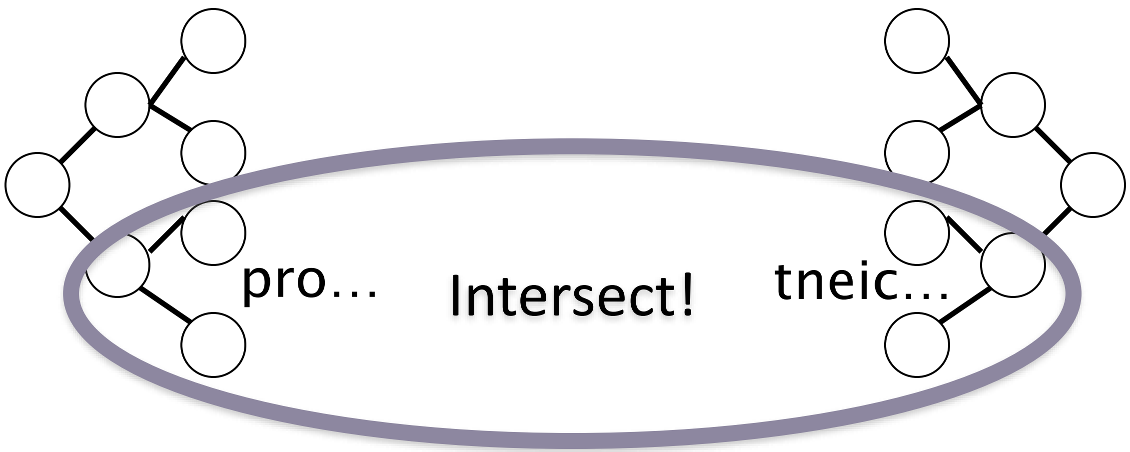
- ***mon:** find docs with words ending in "mon"
  - Maintain an additional tree for terms reversed
  - Retrieve all words in range: ***nom ≤ w < non.***



…
nomel
nomlas
nommoc
…

Handling general wildcard queries

- How about ***pro*cient***?
  - Retrieve possible words for pro* and *cient from the trees and intersect



…
proceed
proficient
profile
…

…
tneiciffe
tneiciffus
tneiciforp
…

Handling general wildcard queries

- General wildcard queries: X*Y
- Look up ***X**** in a normal tree AND ****Y*** in a reverse tree, and then intersect the two term sets
  - Expensive
- The solution: transform wildcard queries into prefix queries (i.e., * occurs at the end)
- This gives rise to the **Permuterm** Index.

Permuterm index

- For the term ***hello***, add an end marker $ and index all rotations:
  - ***hello$, ello$h, llohe, lohel, o$hell*** *and* ***$hello***
- For a wildcard query, add an end marker $ and look up using the rotation with * at the end
  - **X*** lookup on $**X*** ***X** lookup on **X$***
  - **X*Y** lookup on **Y$X*** ***X*** lookup on **X***

Query = hel*o
X=hel, Y=o
Lookup o$hel*

Not so quick Q:
What about X*Y*Z?

Permuterm index

- Lexicon size blows up, proportional to average word length
  - E.g., A 5-letter word, **hello**, has 6 rotations

Is there any other solution?

Bigram (*k*-gram) index

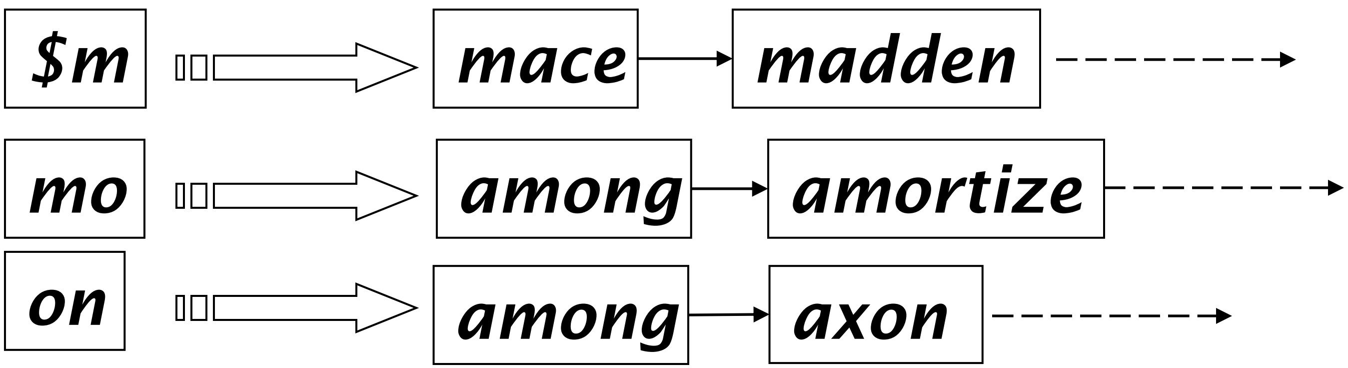
- Enumerate all *k*-grams (sequence of *k* chars) occurring in any term
- *e.g.*, from text "***April is the cruelest month***" we get the 2-grams (*bigrams*)

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

  - As before "$" 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 (here *k*=2).



- Query ***mon**** can now be run as an "AND" Query
  - ***$m*** *AND* ***mo*** *AND* ***on***
  - Possible matches: ***month***, ***moon***, …

Bigram query processing

- Oops! We also included ***moon***, a false positive!
  - It also contains all 3 bigrams **$m, mo, on**
  - 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 wildcard queries

- After getting the possible terms, we still need to execute a Boolean query for each possible term.
- Wildcards can result in expensive query execution (very large disjunctions…)
  - pyth* AND prog*
- If you encourage laziness, people will respond!

Search

Type your search terms, use ‘*’ if you need to.
E.g., Alex* will match Alexander.

Which web search engines allow wildcard queries?

SPELLING CORRECTION

Query misspellings

- Need to correct user queries to retrieve "right" answers
  - E.g., the query ***Ellon Mask***
- We can
  - Return several suggested alternative queries with the correct spelling
    - *"Did you mean … ?"*
  - Retrieve documents indexed by the correct spelling

Spellling corektion

- 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> Narita.***

Fundamental premise

- There is a lexicon of correct spellings.

- Two basic choices for this
  - A standard lexicon such as
    - Merriam-Webster's English Dictionary
    - A domain-specific lexicon – often hand-maintained
  - The lexicon of the indexed corpus
    - E.g., all words on the web
    - All names, acronyms, etc. (including misspellings)

Isolated word correction

- Given a lexicon and a character sequence Q, return the words in the lexicon closest to Q
  - dof → dog, dock, cat....?
- How do we define "closest"?
- We'll study two alternatives
  1. Edit distance (Levenshtein distance)
  2. ngram overlap

1. Edit distance

- Given two strings S_1 and S_2, the edit distance D (S_1, S_2) is the minimum number of operations to convert one to the other
- Operations are typically character-level
  - Insert, Delete, Replace
- E.g., D (**dof** , **dog**) = 1
  - D (**cat**, **act**) = 2.
  - D (**cat**, **dog**) = 3.
- Generally found by dynamic programming

Dynamic Programming

Not dynamic and *not* programming

- Build up solutions of "simpler" instances from small to large
  - Compute solutions of "simpler" instances
  - Use these solutions to solve larger problems
  - E.g., Fibonacci numbers

Fib(1)	Fib(2)	Fib(3)	Fib(4)	Fib(5)
1	1	1+1=2	1+2=3	2+3=5

- Useful when problem can be solved using solution of two or more instances that are only slightly simpler than original instances

Computing Edit Distance

- Let's try to compute the edit distance between S_1 = **PAT** and S_2 = **APT** using this array E, where
  - E (i, j) = the distance between S_1 (up to the i-th character) and S_2 (up to the j-th character)
  - "_" denotes an empty string

		0	1	2	3
	S_2 \ S_1	**_**	**P**	**A**	**T**
0	**_**				
1	**A**				
2	**P**				
3	**T**				

- E (0, 0) = D (_, _)
- E (1, 2) = D (P, AP)
- E (3, 3) = D (PAT, APT)

Computing Edit Distance

- E.g., base cases
  - D (_, _) = 0
  - D (P, _) = 1
  - D (_, A) = 1

		0	1	2	3
	S_1 / S_2	_	P	A	T
0	_	0	1		
1	A	1			
2	P				
3	T				

Computing Edit Distance

- E.g., recursive cases
  - D (PAT, APT) = ??
- What are the smaller problems?
  - If we know D (PAT, AP), the final distance is D (PAT, AP) + 1 since we need **one insertion** to add T to the end of **AP**.
  - If we know D (PA, APT), the final distance is D (PA, APT) + 1 since we need **one insertion** to add T to the end of **PA**.
  - If we know D (PA, AP), the final distance is D (PA, AP) since **inserting T to both PA and AP** does not change the distance.
- What is the minimal distance?

Computing Edit Distance

D(PAT, APT) @ E (3, 3) = min {

D(PAT, AP) @ E(3, 2) + 1,

D(PA, APT) @ E(2, 3) + 1,

D(PA, AP) @ E(2, 2) + 0

} = 2

		0	1	2	3
	S_2 \ S_1	_	**P**	**A**	**T**
0	_	0	1	2	3
1	**A**	1	1	1	2
2	**P**	2	1	2	2
3	**T**	3	2	2	2

$$E(i, j) = \min\{\; E(i, j-1) + 1,\; E(i-1, j) + 1,\; E(i-1, j-1) + \mathbf{m}\}$$

where $\mathbf{m}$ = **1** if $P_i \neq T_j$, **0** otherwise

Edit distance to all dictionary terms?

- Given a (misspelled) 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

2. Ngram overlap

- Enumerate all the ngrams in the query string as well as in the lexicon
  - Query term: **lord** → Bigrams: {lo, or, rd}
  - Lexicon term: **lore** → Bigrams {lo, or, re}
  - Lexicon term: **border** → Bigrams {bo, or, rd, de, er}
- Count the overlaps between a pair of terms
  - 2 between lord and lore
  - 2 between lord and border

This favors longer terms by nature, why?

- Threshold to decide if you have a match
  - E.g., if count >= 2, declare a match

A normalized option – Jaccard coefficient

- Let *X* and *Y* be two sets; then the J.C. is

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

A generally useful overlap measure, even outside of IR

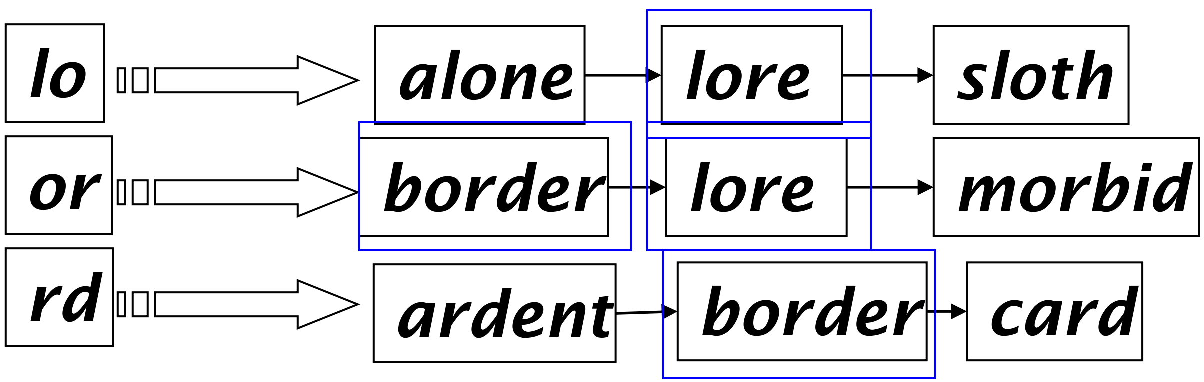
  - Equals 1 when *X* and *Y* have the same elements and 0 when they are disjoint
  - Does not favor longer terms.
  - E.g., JC(lord, lore) = 2/4
    JC(lord, border) = 2/6

"*coefficient de communauté*"

- Threshold to decide if you have a match
  - E.g., if Jaccard >= 0.5, declare a match

Matching bigrams

- Index the dictionary terms using bigram.
- Identify words with at least 2 overlaps (and Jaccard >= 0.5) by merging.



Standard postings "merge" enumerates terms with multiple overlaps

Context-sensitive correction

- **Query**: flew form Narita
- Need context to correct "form" to "from"

- Retrieve dictionary terms close (e.g., in edit distance) to each query term

- Enumerate all possible resulting phrases with one word "corrected" at a time
  - *flew* ***from*** *Narita*
  - ***fled*** *form Narita*
  - *flew form* ***Arita***

Which one to pick?

Context-sensitive correction

- Decide which ones to present using heuristics
  - **Hit-based spelling correction**
    - The correction with most hits

  - E.g., *flew* ***from*** *Narita* (100,000 hits) ← pick this!

    fled *form Narita* (200 hits)

    flew form ***Arita*** (500 hits)

General issues in spelling correction

- Confirm with the user vs. search automatically (e.g., with the most possible correction)
  - Disempowerment or effort saved?
- High computational cost
  - Avoid running routinely on every query?
  - Run only on queries that matched few docs

SOUNDEX

Blanks on slides, you may want to fill in

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
- Available in most databases (Oracle, Microsoft, …)
- We'll explore this just in the context of English

To think about: what other languages does it make sense for?

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)

- See Wikipedia's entry: https://en.wikipedia.org/wiki/Soundex

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

Herman
1. Herman
2. H0rm0n
3. H06505

…

Soundex continued

4. Repeatedly remove one out of each pair of consecutive identical 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.

…
3. H06505
4. H06505
5. H655
6. H655

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

How useful is Soundex?

- Not very – for general IR, spelling correction
- Okay for "high recall" tasks (e.g., Interpol), though biased to names of certain nationalities
  - Sucks for Chinese names: Xin (Pinyin) and Hsin (Wade-Giles) mapped completely different
- Might be more useful with Voice Input

Now what queries can we process?

- We have
  - Positional inverted index with skip pointers
  - Wildcard index
  - Spelling correction
  - Soundex
- Queries such as

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

Summary

- Data Structures for the Dictionary
  - Hash
  - Trees

- Learning to be tolerant

1. Wildcards
   - General Trees
   - Permuterm
   - Ngrams, redux
2. Spelling Correction
   - Edit Distance
   - Ngrams, re-redux
3. Phonetic – Soundex

Resources

- IIR 3, MG 4.2
- Efficient spelling retrieval:
  - K. Kukich. Techniques for automatically correcting words in text. ACM Computing Surveys 24(4), Dec 1992.
  - J. Zobel and P. Dart. Finding approximate matches in large lexicons. Software - practice and experience 25(3), March 1995. http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.14.3856&rep=rep1&type=pdf
  - Mikael Tillenius: Efficient Generation and Ranking of Spelling Error Corrections. Master's thesis at Sweden's Royal Institute of Technology. http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.49.1392
- **Nice, easy reading on spelling correction:**
  - Peter Norvig: How to write a spelling corrector

http://norvig.com/spell-correct.html

It's in python!