AI6122 Text Data Management & Analysis

Topic: Tolerant Retrieval

This lecture

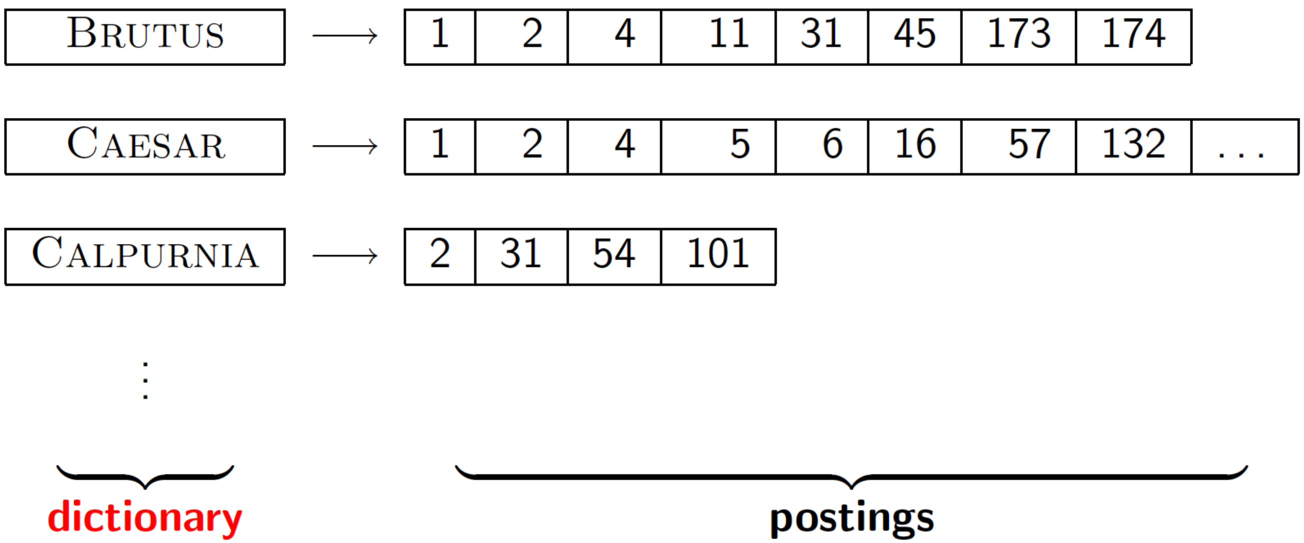
- **Dictionary** data structures

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

Dictionary data structures for inverted indexes

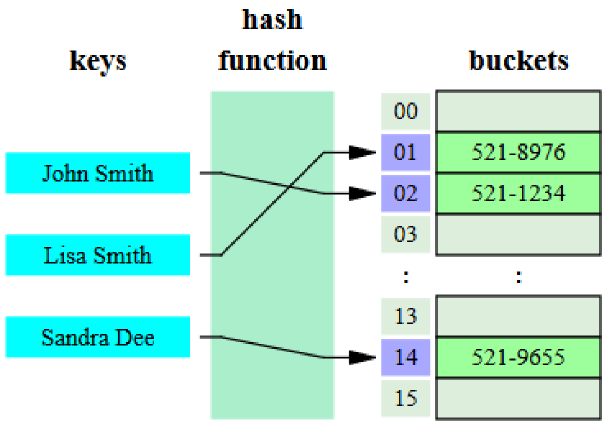
- The dictionary data structure stores the term vocabulary, document frequency, pointers to each postings list …

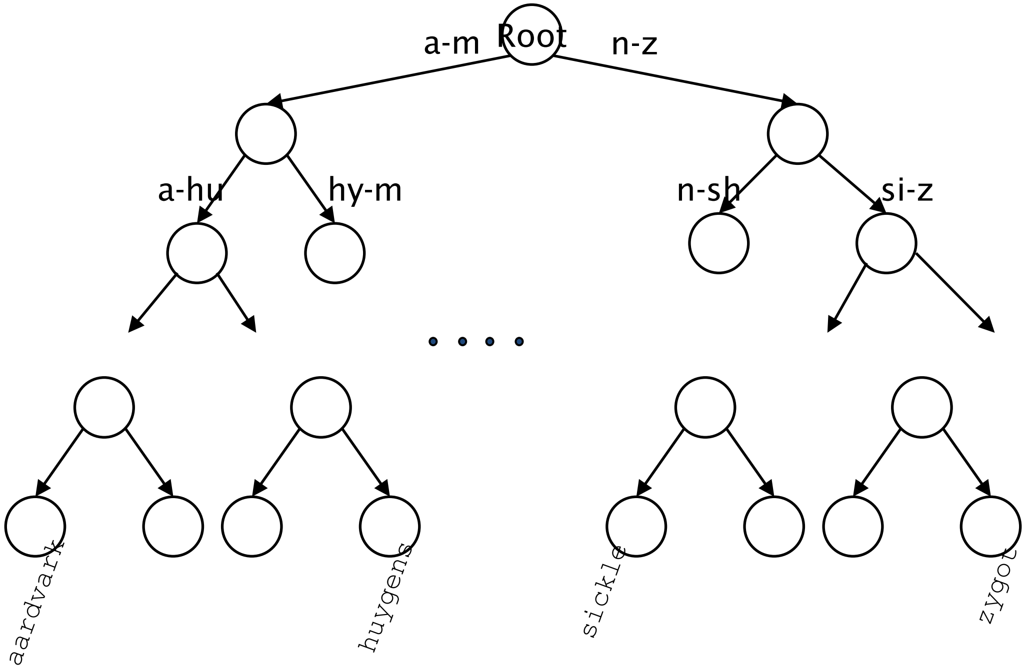
- **In what data structure?**



Dictionary data structures

- Two main choices:
  - Hash table
  - Tree





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*

Trees

- Simplest: binary tree, or more widely used: B-trees
  - Every internal node has a number of children in the interval [a, b] where a, b are appropriate natural numbers, e.g., [2,4].

- Trees require a standard ordering of characters and hence strings

- Pros:
  - Solves the prefix problem (terms starting with hyp)

- Cons:
  - Slower than hash
  - Rebalancing binary trees is expensive

“Tolerant” retrieval

- Wild-card queries
  - **mon***: find all docs containing any word beginning “mon”.
- Spelling correction
  - Isolated word
  - Context-sensitive
- Soundex
  - Words with similar pronunciation

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.

- How about ***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.

- Consider an example 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

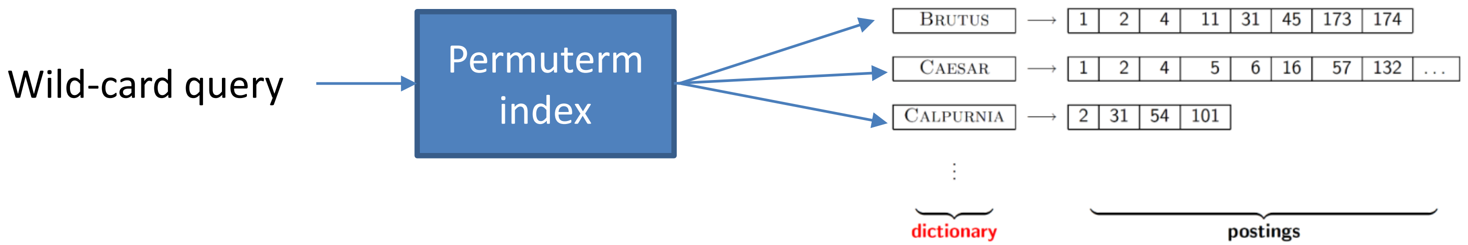
- How can we handle *'s in the middle of query term?
  - ***co*tion***

- We could look up ***co**** AND ***tion** in B-tree and intersect the two term sets
  - Expensive

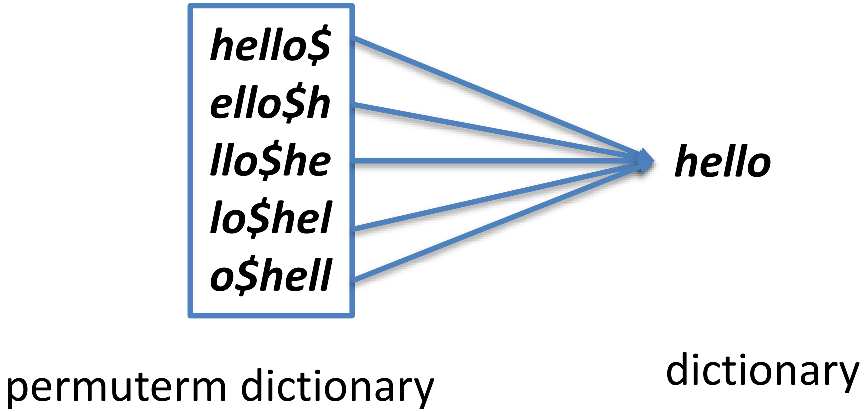
- The solution:
  - Transform wild-card queries so that the *'s occur at the end
  - This gives rise to the **Permuterm** Index.

Permuterm index

- Permuterm index is an index of "the terms in the vocabulary"



- For term ***hello***, index under: ***hello$, ello$h, llohe, lohel, o$hell***
  – Symbol $ is a special symbol, to mark the end of a term

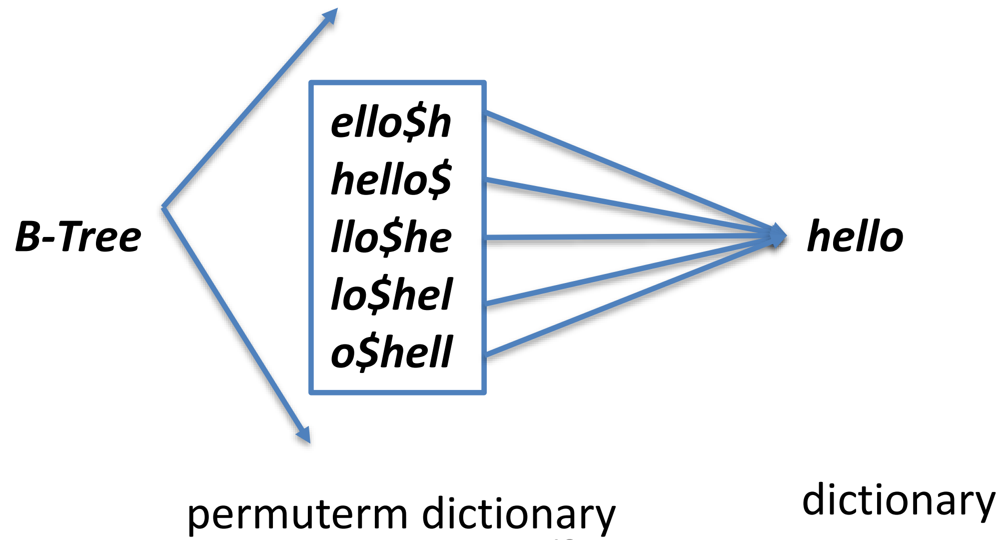


Permuterm index: Examples

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

- Queries:
  - **X** lookup on **X$** **X*** lookup on **X***
  - ***X** lookup on **X$*** ***X*** lookup on **X***
  - **X*Y** lookup on **Y$X***

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



Permuterm query processing

- Rotate query wild-card to the right
- Now use B-tree lookup as before –
  - the terms in the permuterm index are sorted
- Permuterm problem: ≈ quadruples lexicon size
  - Empirical observation for English.
- Alternative approach?
  - Bigram (*k*-gram) indexes

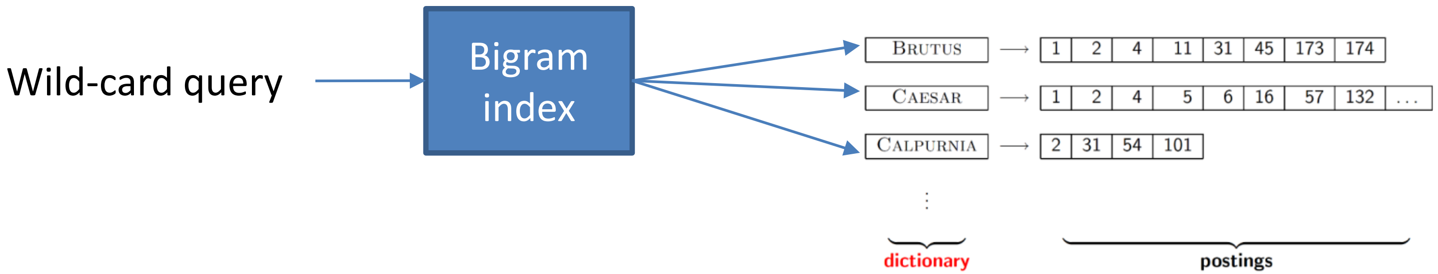
Bigram (*k*-gram) indexes

- Enumerate all *k*-grams (sequence of *k* chars) occurring in any term,
  - From the term “April” we get the 2-grams (*bigrams*)

a, ap, pr, ri, il, l

  - $ 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).

$m → mace → madden

mo → among → amortize

on → among → arond

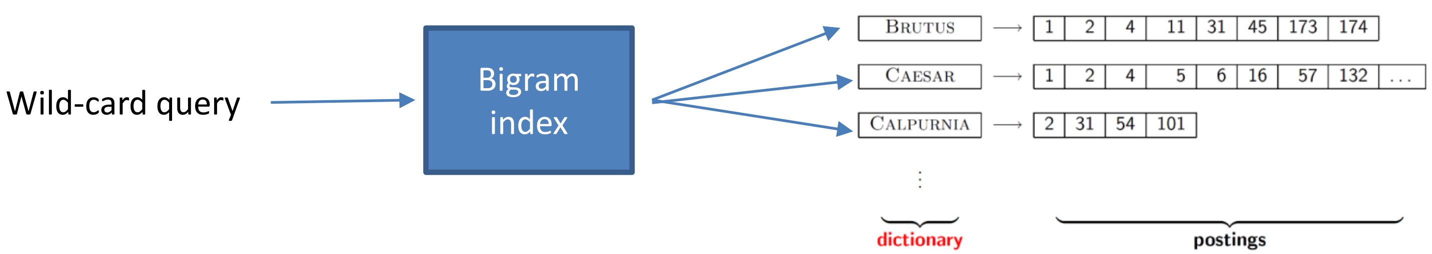
Processing wild-cards

- 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…e.g., pyth* AND prog*



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*** $\rightarrow$ ***form***

  - Context-sensitive
    - Look at surrounding words,
    - e.g., ***I flew <u>form</u> Heathrow to Narita.***

Document correction

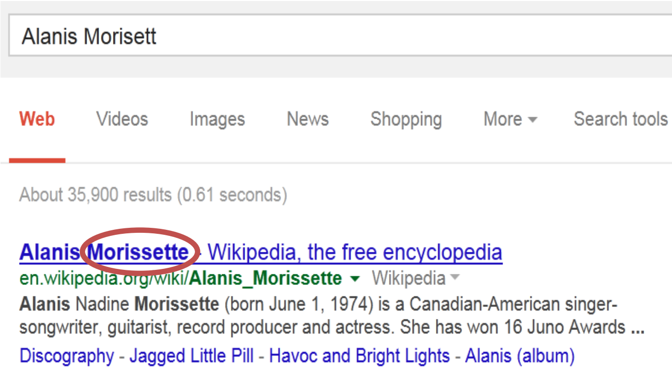
- Especially needed for OCR'ed documents
  - Correction algorithms are tuned for this: rn/m
  - Can use domain-specific knowledge
    - E.g., OCR can confuse O and D more often than it would confuse O and I (adjacent on the QWERTY keyboard, so more likely interchanged in typing).

- 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**"?
  - Edit distance (Levenshtein distance)
  - Weighted edit distance
  - *n*-gram overlap

Edit distance and weighted edit distance

- 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, Substitute

- Weight edit distance: 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,
  - 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 disempower 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**
  - 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:
  - ***november*** trigram***:*** *nov, ove, vem, emb, mbe, ber*.
  - ***december*** trigram***:*** *dec, ece, cem, emb, mbe, ber*.
  - tri-grams overlap: 3
  - Alternative measure: Jaccard coefficient

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 to each query term (in weighted edit distance)
  - 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**.

Another approach

- Break phrase query into a conjunction of biwords
  - ***flew form AND form Heathrow***
- Look for biwords that need only one term corrected.
  - ***flew **** **** form*** ***form **** * ***Heathrow***
- 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 for especially popular, topical queries

- Spell-correction is **computationally expensive**
  - Avoid running routinely on every query?
  - Run only on queries that matched few docs

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

- Step 1: Retain the first letter of the word.
- Step2: Change all occurrences of the following letters to '0' (zero):
  - 'A', E', 'I', 'O', 'U', 'H', 'W', 'Y'.
- Step 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

- Step4: Remove all pairs of consecutive digits.
- Step5: Remove all zeros from the resulting string.
- Step 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

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)***