Chapter #3 Dictionaries & Tolerant Retrieval

★Data Structures for Dictionary=>

↳ **Hashtables =>**

↳ Every vocab is hashed to an integer.

↳ Lookup is fast O(1)

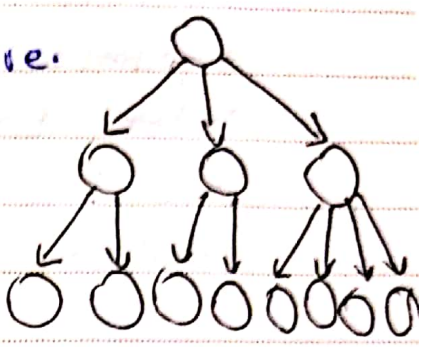
↳ Problems =>

↳ No easy way to find variants.

↳ No prefix search.

↳ may need to rehash over growing vocabulary.

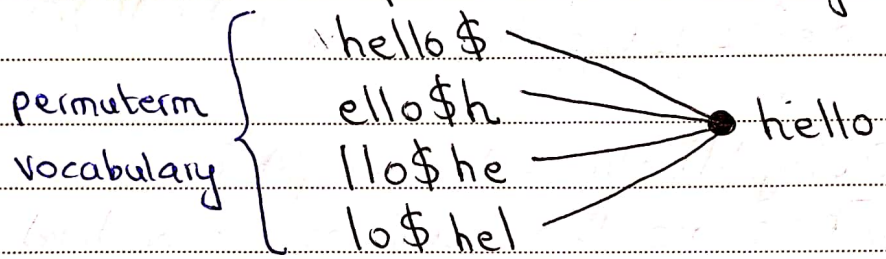
↳ Trees =>
  ↳ Simplest => Binary Trees
    ↳ Rebalancing binary trees is expensive.
  ↳ Usual Approach => B-Trees
    ↳ mitigate the rebalancing problem
  ↳ Pros =>
    ↳ solves the prefix retrieval problem
  ↳ Cons =>
    ↳ Slower O(log M) {M is the height}



* Wildcard Queries => There are two situations for wildcard queries:
  ↳ user is uncertain of the spelling
  ↳ user is aware of multiple variants & seeking all.
  ↳ Trailing wildcards =>
    ↳ Queries are of the form "mon*"
      ↳ find all documents containing words starting with mon.
      ↳ easy with binary & b-trees.
  ↳ Leading wildcards =>
    ↳ Queries are of the form "*mon"
      ↳ find all documents containing words ending with mon.
      ↳ harder since a reverse B-tree is required additionally.
  ↳ center wildcards =>
    ↳ Queries of the form "co*tion"
      ↳ first lookup "co*", then "*tion" in a B-tree and then intersect the two sets.
      ↳ It is extremely expensive.
  ↳ Techniques to handle general wildcards => The strategy is to express the wildcard query q/w as a Boolean query Q on a specially constructed index such that the answer to Q is a superset of the set of vocabs matching q/w. Then, we check each term in Q against q/w, discarding mismatches, then resort to inverted indexes.

Dated:

↳ Permuterm Indexes ⇒ Firstly, we introduce a new symbol $ which marks the end of the term. e.g: hello → hello$. Next, construct a permuterm index in which various rotations of each term all link to the original vocabulary term. We call the rotations as permuterm vocabulary.



↳ How are queries looked up?

Consider the query "m*n", firstly rotate the query to turn this into a trailing wildcard query e.g: "n$m*". Next, we lookup this string into a permuterm index. Finally, we can lookup in B-Trees for finding the original vocabulary like "man" and "moron". Finally, we can use inverted index to retrieve the results.

Wildcard query → rotate → permuterm index → B-Tree → Inverted Index

↳ Multiple wildcards ⇒

For a query such as "fi*mo*er". Firstly, we enumerate the terms involving "er$fi*". Not all dictionary terms will have "mo" in the middle, so we filter such terms by "exhaustive enumeration", terms like fishmonger would survive but fillbuster will not. Then inverted index can be used to find the results.

↳ Problems ⇒

↳ Quadruples Lexicon size due to permutative rotations.

efficiency ++ Dated:

↳ Bigram (k-gram) indexes => In the k-gram indexes, the dictionary contains all k-grams that occur in any term in the vocab. Each posting list points from a k-gram to the terms containing that k-gram.
e.g: "cruelest month" will have the bigrams → "$c, cr, ru, ue, el, le, es, st, t$, m, mo, on, nt, th, h"

$m → mace → madden → ...

mo → among → amortize → ...

↳ How are queries looked up? → always keep * at end
Consider the query "mon*", the query can now be run as "$m AND mo AND on". But we'd end up enumerating "moon" as well. So a post-filtering step is introduced so string compare the results with the original query. Survivors are checked in inverted indexes.

★ Spelling Corrections =>

↳ Principle uses

↳ correcting documents being indexed

↳ ~~will/not/catch typos~~ correcting user queries

↳ Isolated Word Correction => Given a lexicon and a character sequence Q, return the words in lexicon closest to Q. We don't care about context and words originally in correct spellings won't be fixed. The following strategies can be used to implement this:

↳ Levenshtein Distance =>

Given two strings S_1 and S_2, the levenshtein distance is the minimum number of operations to convert one to the other. The operations are: Insert, Replace, Delete.

e.g => dof → dog cost = 1

cat → act cost = 2

Dated: _____

len(s1) = i, len(s2) = j

↳↳↳ EDIT DISTANCE (s1, s2):

```
1   int m[i][j] = {0}
2   for i ← 1 to |s1|      } s2 is empty so increment
3       m[i, 0] = i        }
4   for j ← 1 to |s2|      } s1 is empty so increment
5       m[0, j] = j        }
6   for i ← 1 to |s1|:          diagonal+0 if same
7       for j ← 1 to |s2|:      diagonal+1 if different
8           m[i,j] = min{m[i-1,j-1] + s1[i]=s2[j] ? 0 : 1,
9                       m[i-1, j]+1,   } above+1
10                      m[i, j-1] +1}  } left+1
11  return m[|s1|, |s2|]  } last index is the solution
```

ex:

	∅	f	a	s	t	
∅	0	1	2	3	4	∅
c	1	1	2	3	4	c
a	2	2	1	2	3	a
t	3	3	2	3	2	t
s	4	4	3	2	3	s

cost = 3

↳ How to use?

↳ The simplest idea is to find all strings with minimum edit distance and then return them but this is extremly expensive.

One efficient heuristic is to restrict the search dictionary terms beginning with the same letter assuming the first letter is always correct.

A sophisticated variant is to use permuterm index while omitting $. We can also refine the rotation scheme by omitting 'l' characters before each rotation so to have best matches. Then, perform B-Tree traversal.

Dated: _____

↳↳↳ k-gram overlaps ⇒ We can improve the use of edit distance with k-gram indexes. The method is to use k-gram indexes to find the relevant k-grams then use edit distances over them to find the most relevant. Consider bi-grams of lord:

lo → alone → lord → sloth

or → border → lord → morbid

rd → ardent → border → card

then merging can be performed to retrieve relevants.

Linear merging is still troublesome since for queries like "bord" will retrieve "boardroom", so a measure called "Jaccard Coefficient" is adapted. This is a measure of similarity b/w two terms. It is the measure of overlap $|A \cap B| / |A \cup B|$.

To compute the ~~k-gram~~ co-eff, we need to find the k-grams of query and terms. Consider, q1 = bord and t = boardroom, the jaccard co-efficient can be calculated by dividing the number of k-gram hits by the sum of total bigrams − posting hits which is ⇒ {bo, or, rd} ⇒ $\frac{2}{8+3-2} = 0.222$.

{bo, oa, ar, rd, dr, ro, oo, om}

Finally, we can check if the jaccard co-efficient exceeds a certain threshold, we add the term to the answer.

↳ Context Sensitive Spelling Correction ⇒

Isolated words will fail to solve typographical errors when all terms are spelled correctly such as: "flew form Heathrow". We can try all permutations for each term but that would be computationally expensive. The other heuristics are to find hit-based spelling corrections and then among

the retrieved results we can use the alternative with most hitting docs or by query log analysis to find relevant user queries to this query.

↳ Phonetic Correction ⇒

The main idea is to generate a phonetic hash so that similar sounding terms are always together.

↳ Soundex Algorithm ⇒

This algorithm is used to generate the phonetic hashes.

↳ Scheme ⇒

↳ Turn every term into a 4-character reduced form, then build an inverted index from these reduced forms to original words called the Soundex Index.

↳ Do the same with query terms.

↳ Run the queries over the soundex index.

↳ Encoding ⇒ The first character is the first letter of the term, rest are digits from 0-9.

↳ How to convert?

↳ Retain the first letter of the term.

↳ change all occurences of A, E, I, O, U, H, Y, W to 0.

↳ change following

B, F, P, V → 1

C, G, J, K, Q, S, X, Z → 2

D, T → 3

L → 4

M, N → 5

R → 6

↳ Remove all pairs of consecutive digits.

Dated: _____

↳ Remove all zeros
↳ Pad trailing zeros and return the first four characters.

↳ Query Processing
↳ A query like "Hermann" is processed like
Hermann → HOrmOnn → H065055 → H6555
return H655[first four]

↳ Observations
↳ vowels are viewed as interchangeable in transcribing names.
↳ Consonants with similar sounds are put in equivalence classes.
↳ While it works to some extent, it is accurate to some extent but not completely
e.g: Rafi, Rafee have different soundex codes.

↳ Limitations
↳ It can handle only English words.
↳ It gives false +ves and false -ves.
↳ It does not consider semantic meaning.