Introduction to Information Retrieval

Lecture 4: Dictionaries and tolerant retrieval
This lecture

- Dictionary data structures
- “Tolerant” retrieval
  - Wild-card queries
  - Spelling correction
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**: 1 2 4 11 31 45 173 174
  - **Caesar**: 1 2 4 5 6 16 57 132 ...
  - **Calpurnia**: 2 31 54 101

...
A naïve dictionary

- An array of struct:

<table>
<thead>
<tr>
<th>term</th>
<th>document frequency</th>
<th>pointer to postings list</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td>65,265</td>
<td></td>
</tr>
<tr>
<td>aachen</td>
<td>65</td>
<td></td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>zulu</td>
<td>221</td>
<td></td>
</tr>
</tbody>
</table>

- 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 hashables 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

Root

a-m

n-z

a-hu

hy-m

n-sh

si-z

aardvark

huygens

sickle

zygot
Tree: B-tree

- Definition: 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

- 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: \( \text{mon} \leq w < \text{moo} \)
- **mon**: find words ending in “mon”: harder
  - Maintain an additional B-tree for terms *backwards*.
  - Can retrieve all words in range: \( \text{nom} \leq w < \text{non} \).

Exercise: from this, how can we enumerate all terms meeting the wild-card query \textit{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
  - Still need verification to remove false-positives

- 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, llo$he, lo$hel, o$hell`
  where $ is a special symbol.

- Queries:
  - **P** Exact match `P$
  - **P** Range match `$P$
  - **P** Range match `P$
  - **P**Q Range match `Q$P$
  - **P**Q*R ??? Exercise!

Q: Why not `P**`?

Query = `hel*o`
P=hel, Q=o
Lookup `o$hel*`
Permuterm query processing

- Rotate query wild-card to the right
- Now use B-tree lookup as before.
- **Permuterm problem:** $\approx$ 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 (here $k=2$).

- $m \rightarrow mace \rightarrow madden$
- $mo \rightarrow among \rightarrow amortize$
- $on \rightarrow among \rightarrow mound$
Processing wild-cards

- Query $\text{mon}^*$ can now be run as $\text{sm AND mo AND on}$
- Gets terms that match AND version of our wildcard query.
- But we’d enumerate $\text{moon}$.
- Must verify 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!

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

- Efficient spell retrieval:
    [http://citeseer.ist.psu.edu/zobel95finding.html](http://citeseer.ist.psu.edu/zobel95finding.html)
    [http://citeseer.ist.psu.edu/179155.html](http://citeseer.ist.psu.edu/179155.html)

- Nice, easy reading on spell correction:
  - Peter Norvig: How to write a spelling corrector