Chapter 3

Dictionaries and Tolerant Retrieval
Dictionaries

- **Dictionary**: the data structure for storing the term vocabulary
- **Term vocabulary**: the data

<table>
<thead>
<tr>
<th>Term</th>
<th>postings</th>
</tr>
</thead>
<tbody>
<tr>
<td>Brutus</td>
<td>1 2 4 11 31 45 173 174</td>
</tr>
<tr>
<td>Caesar</td>
<td>1 2 4 5 6 16 57 132 ...</td>
</tr>
<tr>
<td>Calpurnia</td>
<td>2 31 54 101</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>
For each term, we need to store a couple of items:
  - document frequency
  - pointer to postings list
  - ...

Assume for the time being that we can store this information in a fixed-length entry.

Assume that we store these entries in an array.
Storing Dictionaries (2)

<table>
<thead>
<tr>
<th>term</th>
<th>document frequency</th>
<th>pointer to postings list</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td>656,265</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>

space needed: 20 bytes 4 bytes 4 bytes

- How do we look up an element in this array at query time?
- Remember: these dictionaries can be huge, scanning is not an option
Data Structures

- Two main classes of data structures: hash tables and trees
- Some IR systems use hash tables, some use trees.
- Criteria for when to use hash tables vs. trees:
  - Is there a fixed number of terms or will it keep growing?
  - What are the relative frequencies with which various keys will be accessed?
  - How many terms are we likely to have?
Hash Tables

- Each vocabulary term is hashed into an integer.
- Try to avoid collisions
- At query time, do the following: hash query term, resolve collisions, locate entry in fixed-width array

**Pros:**
- Lookup in a hash table is faster than lookup in a tree.

**Cons:**
- no prefix search (all terms starting with *automat*)
- need to rehash everything periodically if vocabulary keeps growing
Trees

- Trees solve the prefix problem (find all terms starting with *automat*).
- Simplest tree: binary tree
- Binary trees are problematic:
  - Only balanced trees allow efficient retrieval
  - Rebalancing binary trees is expensive
- Use B-trees (index structure you know from database lecture)
B-Tree

Taken from documentation for Oracle 10g
Wildcard Queries

- mon*: find all docs containing any term beginning with mon
- Easy with B-tree dictionary: retrieve all terms \( t \) in the range: \( \text{mon} \leq t < \text{moo} \)
- *mon: find all docs containing any term ending with mon
  - Maintain an additional tree for terms backwards
  - Then retrieve all terms \( t \) in the range: \( \text{nom} \leq t < \text{non} \)
Query Processing

- At this point, we have an enumeration of all terms in the dictionary that match the wildcard query.
- We still have to look up the postings for each enumerated term.
- E.g., consider the query: `gen* AND universit*`
- This may result in the execution of many Boolean AND queries.
Wildcards in Middle of Term

- Example: \textit{m\textstar nchen}

- We could look up \textit{m\textstar} and \textit{\textstar nchen} in the B-tree and intersect the two term sets.

- Expensive (there are probably thousands and thousands of terms beginning with “m”)

- Alternative: \textit{permuterm} index

- Basic idea: Rotate every wildcard query, so that the * occurs at the end.
Permuterm Index

For term hello: add hello$, ello$h, llo$he, lo$hel, and o$hell to the B-tree where $ is a special symbol.
Permuterm Index (2)

- For hello, we’ve stored: hello$, ello$h, llo$he, lo$hel, and o$hell

- Queries
  - For X, look up X$
  - For X*, look up X*$
  - For *X, look up X$
  - For *X*, look up X*
  - For X*Y, look up Y$X*
  - Example: For hel*o, look up o$hel*

- It’s really a tree and should be called permuterm tree.
- But permuterm index is more common name.
Query Processing

- Once we modified the query (as shown on last slide), we can do a regular lookup on a B-tree.
- This is much faster than looking up $X^*$ and $^*Y$ and combining results (for query $X^*Y$).
- Permuterm index also handles leading wildcards: $^*X$.
- It has a disadvantage, though: quadruples the size of the dictionary compared to a regular B-tree (as every term is stored multiple times).
K-gram Index

- More space-efficient than permuterm index
- Enumerate all character $k$-grams (sequence of $k$ characters) occurring in a term
- 2-grams are called *bigrams*.
- Example: from *April is the cruelest month* we get the 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 an inverted index from bigrams to the terms that contain the bigram
Postings List in a 3-gram Index

Note that we now have two different types of inverted indexes

- The term-document inverted index for finding documents based on a query consisting of terms
- The $k$-gram index for finding terms based on a query consisting of $k$-grams
Processing Wildcard Queries

- Query mon* can now be run as:
  $m \text{ AND } mo \text{ AND } on$
- Gets us all terms with the prefix mon . . .
- . . . but also many “false positives” like moon.
- We must postfilter these terms against query.
- Surviving terms are then looked up in the term-document inverted index.
- k-gram indexes are fast and space efficient (compared to permuterm indexes).
Processing Wildcard Queries (2)

- We must potentially execute a large number of Boolean queries for each enumerated, filtered term (on the term-document index)
- Recall the query: `gen* AND universit*`
- Most straightforward semantics: Conjunction of disjunctions
- Very expensive
- Users hate to type.
- If abbreviated queries like `pyth* theo*` for `pythagoras’ theorem` are legal, users will use them
  ...
  ...
  a lot
Spelling Correction

- Two principal uses
  - Correcting documents being indexed
  - Correcting user queries

- Two different methods for spelling correction
  - *Isolated word* spelling correction
    - Check each word on its own for misspelling
    - Will not catch typos resulting in correctly spelled words, e.g., *an asteroid that fell form the sky*

  - *Context-sensitive* spelling correction
    - Look at surrounding words
    - Can correct *form/from* error above
Correcting Documents

- We’re not interested in interactive spelling correction of documents (e.g., MS Word) in this class.
- In IR, we use document correction primarily for OCR’ed documents (i.e. documents digitized via Optical Character Recognition)
- The general philosophy in IR is: don’t change the documents.
Correcting Queries

- First: isolated word spelling correction
- Fundamental premise 1: There is a list of “correct words” from which the correct spellings come.
- Fundamental premise 2: We have a way of computing the distance between a misspelled word and a correct word.
- Simple spelling correction algorithm: return the “correct” word that has the smallest distance to the misspelled word.
- Example: *informaton* $\rightarrow$ *information*
Correcting Queries (2)

- Can we use the term vocabulary of the inverted index as the list of correct words?
  - Term vocabulary can be very biased
  - May be missing certain terms

- Alternatives:
  - A standard dictionary (Webster’s, Encyclopædia Britannica, etc.)
  - An industry-specific dictionary (for specialized IR systems)
  - The term vocabulary of the collection, appropriately weighted
Computing Distance

- How can we compute the distance between words?
- We’ll look at some alternatives:
  - Edit distance (Levenshtein distance)
  - Weighted edit distance
  - $k$-gram overlap
Edit Distance

The (minimum) edit distance between string $s_1$ and string $s_2$ is the minimum number of basic operations to convert $s_1$ to $s_2$.

Levenshtein distance: The admissible basic operations are insert, delete, and replace

Levenshtein distance $dog$-$do$: 1 (deletion)

Levenshtein distance $cat$-$cart$: 1 (insertion)

Levenshtein distance $cat$-$cut$: 1 (replacement)

Levenshtein distance $cat$-$act$: 2
(2 replacements or 1 insertion and 1 deletion)
Computing Distance

- Getting from *cats* to *fast*

<table>
<thead>
<tr>
<th></th>
<th>“”</th>
<th>f</th>
<th>a</th>
<th>s</th>
<th>t</th>
</tr>
</thead>
<tbody>
<tr>
<td>“”</td>
<td>“” → “”</td>
<td>“” → f</td>
<td>“” → fa</td>
<td>“” → fas</td>
<td>“” → fast</td>
</tr>
<tr>
<td>c</td>
<td>c → “”</td>
<td>c → f</td>
<td>c → fa</td>
<td>c → fas</td>
<td>c → fast</td>
</tr>
<tr>
<td>a</td>
<td>ca → “”</td>
<td>ca → f</td>
<td>ca → fa</td>
<td>ca → fas</td>
<td>ca → fast</td>
</tr>
<tr>
<td>t</td>
<td>cat → “”</td>
<td>cat → f</td>
<td>cat → fa</td>
<td>cat → fas</td>
<td>cat → fast</td>
</tr>
<tr>
<td>s</td>
<td>cats → “”</td>
<td>cats → f</td>
<td>cats → fa</td>
<td>cats → fas</td>
<td>cats → fast</td>
</tr>
</tbody>
</table>

- Each cell will contain the (cheapest) costs of getting from the string on the left-hand side to the string on the right-hand side
We know the costs for the uppermost row and the leftmost column:

- we have to get from "" to *fast* by inserting characters
- we have to get from *cats* to "" by deleting characters
Other Cells

- Coming from (a): add 1 to cost in (a) (insertion)
- Coming from (b): add 1 to cost in (b) (deletion)
- Coming from (c):
  - if characters in row and column are equal, copy costs from (c)
  - if letters are not equal, add 1 to cost in (c) (replacement)
- Take minimum of costs
Computing the costs for all cells results in the following matrix:

<table>
<thead>
<tr>
<th></th>
<th>“”</th>
<th>f</th>
<th>a</th>
<th>s</th>
<th>t</th>
</tr>
</thead>
<tbody>
<tr>
<td>“”</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
</tr>
<tr>
<td>c</td>
<td>1</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
</tr>
<tr>
<td>a</td>
<td>2</td>
<td>2</td>
<td>1</td>
<td>2</td>
<td>3</td>
</tr>
<tr>
<td>t</td>
<td>3</td>
<td>3</td>
<td>2</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>s</td>
<td>4</td>
<td>4</td>
<td>3</td>
<td>2</td>
<td>3</td>
</tr>
</tbody>
</table>

So the Levenshtein distance is 3
Algorithm

\begin{algorithm}
\textbf{EDITDISTANCE}(s_1, s_2) \\
1 \hspace{1em} \text{int } m[i, j] = 0 \\
2 \hspace{1em} \text{for } i \leftarrow 1 \text{ to } |s_1| \\
3 \hspace{1em} \text{do } m[i, 0] = i \\
4 \hspace{1em} \text{for } j \leftarrow 1 \text{ to } |s_2| \\
5 \hspace{1em} \text{do } m[0, j] = j \\
6 \hspace{1em} \text{for } i \leftarrow 1 \text{ to } |s_1| \\
7 \hspace{1em} \text{do for } j \leftarrow 1 \text{ to } |s_2| \\
8 \hspace{2em} \text{do } m[i, j] = \min\{m[i - 1, j - 1] + \text{if } (s_1[i] = s_2[j]) \text{ then } 0 \text{ else } 1, \\
9 \hspace{2em} m[i - 1, j] + 1, \\
10 \hspace{2em} m[i, j - 1] + 1\} \\
11 \hspace{1em} \text{return } m[|s_1|, |s_2|]
\end{algorithm}
Weighted Edit Distance

As Levenshtein distance, but weight of an operation depends on the characters involved.

Meant to capture keyboard errors, e.g., $m$ more likely to be mistyped as $n$ than as $q$.

Therefore, replacing $m$ by $n$ is a smaller edit distance than by $q$.

We now require a weight matrix as input.

Modify dynamic programming to handle weights.
Using Edit Distances

- Comparing query term $q$ to all terms in the vocabulary is too expensive.

Solution: use heuristics to determine subset

- Only compare to terms beginning with the same letter (doesn’t work for typos at beginning).
- Generate set of rotations for $q$ and use a permuterm index (doesn’t work well for replacements).
- For each rotation, omit a suffix of $l$ characters before doing lookup in permuterm index.
  - Ensures that each term in query rotation shares a substring with retrieved terms.
  - The value of $l$ could be fixed to a constant length (e.g. 2), or depend on the length of $q$. 
Using a K-gram Index

- Enumerate all $k$-grams in the query term
- Use the $k$-gram index to retrieve “correct” words that match query term $k$-grams
- Threshold by number of matching $k$-grams
- E.g., only vocabulary terms that differ by at most 3 $k$-grams
- Example: bigram index, misspelled word *bordroom*
- Bigrams: *bo, or, rd, dr, ro, oo, om*
Example

- bo -> aboard -> about -> boardroom -> border
- or -> border -> lord -> morbid -> sordid
- rd -> aboard -> ardent -> boardroom -> border
Example with 3-grams

Issue: Fixed number of $k$-grams that differ does not work for words of differing length.

Suppose the correct word is \texttt{november}:
\texttt{n, no, nov, ove, vem, emb, ber, er, r}

And the query term is \texttt{december}:
\texttt{d, de, dec, ece, cem, emb, mbe, ber, er, r}

So 5 trigrams overlap (out of 10 in each term)

How can we turn this into a normalized measure of overlap?
Jaccard Coefficient

- A commonly used measure of overlap of two sets
- Let $A$ and $B$ be two sets
- Jaccard coefficient:
  \[
  \frac{|A \cap B|}{|A \cup B|}
  \]
- $A$ and $B$ don’t have to be the same size.
- Always assigns a number between 0 and 1.
- Application to spelling correction: declare a match if the coefficient is, say, $> 0.8$. 
Context-sensitive Correction

- Our example was: an asteroid that fell form the sky
- How can we correct form here?
- One idea: hit-based spelling correction
  - Retrieve “correct” terms close to each query term
  - for flew form munich: flea for flew, from for form, munch for munich
  - Now try all possible resulting phrases as queries with one word “fixed” at a time
  - Try query “flea form munich”
  - Try query “flew from munich”
  - Try query “flew form munch”
  - The correct query “flew from munich” should have the most hits.
The “hit-based” algorithm we just outlined is not very efficient.

Suppose we have 7 alternatives for *flew*, 19 for *form* and 3 for *munich*

Then we have to test $7 \times 19 \times 3$ different variants

More efficient alternative: look at “collection” of queries, not documents

This assumes that we log queries
General Issues

User interface
- automatic vs. suggested correction
- *Did you mean* only works for one suggestion.
- What about multiple possible corrections?
- Tradeoff: simple vs. powerful UI

Cost
- Spelling correction is potentially expensive.
- Avoid running on every query?
- Maybe just on queries that match few documents.
Phonetic Matching

- Soundex is the basis for finding *phonetic* (as opposed to orthographic) alternatives.

- Example: *Chebyshev / Tchebyscheff*

- 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
Soundex Algorithm

1. Retain the first letter of the term.
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 to 1
   - C, G, J, K, Q, S, X, Z to 2
   - D, T to 3
   - L to 4
   - M, N to 5
   - R to 6
4. Repeatedly remove one out of each pair of consecutive identical digits
5. Remove all zeros from the resulting string; pad the resulting string with trailing zeros and return the first four positions, which will consist of a letter followed by three digits
Example

<table>
<thead>
<tr>
<th>Steps</th>
<th>Difficulty</th>
<th>Difference</th>
</tr>
</thead>
<tbody>
<tr>
<td>Steps 1 and 2</td>
<td>d0ff0c0lt0</td>
<td>d0ff0r0nc0</td>
</tr>
<tr>
<td>Step 3</td>
<td>d011020430</td>
<td>d011060520</td>
</tr>
<tr>
<td>Step 4</td>
<td>d01020430</td>
<td>d01060520</td>
</tr>
<tr>
<td>Step 5</td>
<td>d124</td>
<td>d165</td>
</tr>
</tbody>
</table>

- Vowels are viewed as being interchangeable
- Consonants with similar sounds (e.g. D and T) are put in equivalence classes
- Works fairly well for European languages
Summary

In this chapter we looked at

- how to organize a dictionary of an inverted index
- how to do imprecise searches on this dictionary
- handling
  - wildcards
  - spelling mistakes