Text Pre-processing and Faster Query Processing

David Kauchak

cs160

Fall 2009

adapted from:
http://www.stanford.edu/class/cs276/handouts/lecture2-Dictionary.ppt
Administrative

- Everyone have CS lab accounts/access?
- Homework 1
  - Page numbers
  - Due before class next Wed.
  - Popular media question
- Issues with assignment 1?
- Discussion board?
- CS lunch today
Outline for today

- Improvements to basic postings lists
  - Speeding up the merge operation
  - Adding phrase queries and proximity queries

- Text pre-processing
  - tokenizing
  - “all but the kitchen sink” - approaches to token normalization

- Regular expressions in Java (time permitting)
Recall the merge

- Walk through the two lists simultaneously

\[ \text{word1} : 2 \rightarrow 4 \rightarrow 8 \rightarrow 16 \rightarrow 32 \rightarrow 64 \rightarrow 128 \]

\[ \text{word2} : 1 \rightarrow 200 \]

\[ O(\text{length1} + \text{length2}) \]

Can we do better?
Can we augment the data structure?
Augment postings with skip pointers (at indexing time)

- How does this help?
Query processing with **skip pointers**
Query processing with skip pointers

128
2
4
8
41
48
64
128

31
16
1
2
3
8
11
17
21
31

128
31
11
31
Query processing with **skip pointers**

We skip these entries.
Where do we place skips?

- **Tradeoff:**
  - More skips $\rightarrow$ shorter skip spans $\Rightarrow$ more likely to skip. But lots of comparisons to skip pointers. More storage required.
  - Fewer skips $\rightarrow$ few pointer comparison, but then long skip spans $\Rightarrow$ few successful skips
Placing skips

- Simple heuristic: for postings of length $L$, use $\sqrt{L}$ evenly-spaced skip pointers.
  - ignores word distribution

- Are there any downsides to skip lists?
- The I/O cost of loading a bigger postings list can outweigh the gains from quicker in memory merging! (Bahle et al. 2002)
- A lot of what we’ll see in the class are options. Depending on the situation some may help, some may not.
Outline for today

- Improvements to basic postings lists
  - Speeding up the merge operation
  - Adding phrase queries and proximity queries

- Text pre-processing
  - tokenizing
  - “all but the kitchen sink” - approaches to token normalization

- Regular expressions in Java (time permitting)
Phrase queries

- Want to be able to answer queries such as “pomona college”
- “I went to a college in pomona” would not a match
  - The concept of phrase queries has proven easily understood by users
  - Many more queries are implicit phrase queries

How can we modify our existing postings lists to support this?
Positional indexes

- In the postings, store a list of the positions in the document where the term occurred.

\[ \text{word1} \quad 2 \rightarrow 4 \rightarrow 8 \rightarrow 16 \]

\[ \text{word1} \quad 2: \langle 3, 16, 20 \rangle \rightarrow 4: \langle 39 \rangle \rightarrow 8: \langle 4, 18, 40 \rangle \rightarrow 16: \langle 7 \rangle \]

\[ \text{docID: \langle position1, position2, \ldots \rangle} \]
Positional index example

be:
1: \langle 7, 18, 33, 72, 86, 231 \rangle
2: \langle 3, 149 \rangle
4: \langle 17, 191, 291, 430, 434 \rangle
5: \langle 363, 367 \rangle

1. Looking only at the “be” postings list, which document(s) could contain “to be or not to be”?

2. Using both postings list, which document(s) could contain “to be or not to be”?

3. Describe an algorithm that discovers the answer to question 2 (hint: think about our linear “merge” procedure)
Processing a phrase query: “to be”

- Find all documents that have the terms using the “merge” procedure.
- For each of these documents, “merge” the position lists with the positions offset depending on where in the query the word occurs.

be:
4: ⟨17,191,291,430,434⟩

4: ⟨17,191,291,430,434⟩

to:
4: ⟨12,13,429,433,500⟩

4: ⟨13,14,430,434,501⟩
Processing a phrase query: “to be”

- Find all documents that have the terms using the “merge” procedure
- For each of these documents, “merge” the position lists with the positions offset depending on where in the query the word occurs

be:
4: ⟨17,191,291,430,434⟩

4: ⟨17,191,291,430,434⟩

to:
4: ⟨12,13,429,433,500⟩

4: ⟨13,14,430,434,501⟩
What about proximity queries?

- Find “pomona” within k words of “college”
- Similar idea, but a bit more challenging

- Naïve algorithm for merging position lists
  - Assume we have access to a merge with offset exactly i procedure (similar to phrase query matching)
  - for i = 1 to k
    - if merge with offset i matches, return a match
    - if merge with offset -i matches, return a match

- Naïve algorithm is inefficient, but doing it efficiently is a bit tricky
You can compress position values/offsets
Nevertheless, a positional index expands postings storage *substantially*
Nevertheless, a positional index is now standardly used because of the power and usefulness of phrase and proximity queries … whether used explicitly or implicitly in a ranking retrieval system
Positional index size

- What does adding positional information do to the size of our index?
- Need an entry for each occurrence, not just once per document
- Posting size depends on the lengths of the documents
Positional index size

- Average web page has <1000 terms
- SEC filings, books, even some epic poems ... easily 100,000 terms
- Consider a term with frequency 0.1%

<table>
<thead>
<tr>
<th>Document size</th>
<th>Postings</th>
<th>Positional postings</th>
</tr>
</thead>
<tbody>
<tr>
<td>1000</td>
<td>?</td>
<td></td>
</tr>
<tr>
<td>100,000</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Positional index size

- Average web page has <1000 terms
- SEC filings, books, even some epic poems ... easily 100,000 terms
- Consider a term with frequency 0.1%

<table>
<thead>
<tr>
<th>Document size</th>
<th>Postings</th>
<th>Positional postings</th>
</tr>
</thead>
<tbody>
<tr>
<td>1000</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>100,000</td>
<td>?</td>
<td></td>
</tr>
</tbody>
</table>
Positional index size

- Average web page has <1000 terms
- SEC filings, books, even some epic poems ... easily 100,000 terms
- Consider a term with frequency 0.1%

<table>
<thead>
<tr>
<th>Document size</th>
<th>Postings</th>
<th>Positional postings</th>
</tr>
</thead>
<tbody>
<tr>
<td>1000</td>
<td>1</td>
<td>?</td>
</tr>
<tr>
<td>100,000</td>
<td>1</td>
<td></td>
</tr>
</tbody>
</table>
Positional index size

- Average web page has <1000 terms
- SEC filings, books, even some epic poems ... easily 100,000 terms
- Consider a term with frequency 0.1%

<table>
<thead>
<tr>
<th>Document size</th>
<th>Postings</th>
<th>Positional postings</th>
</tr>
</thead>
<tbody>
<tr>
<td>1000</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>100,000</td>
<td>1</td>
<td>?</td>
</tr>
</tbody>
</table>
Positional index size

- Average web page has <1000 terms
- SEC filings, books, even some epic poems ... easily 100,000 terms
- Consider a term with frequency 0.1%

<table>
<thead>
<tr>
<th>Document size</th>
<th>Postings</th>
<th>Positional postings</th>
</tr>
</thead>
<tbody>
<tr>
<td>1000</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>100,000</td>
<td>1</td>
<td>100</td>
</tr>
</tbody>
</table>
Rules of thumb

- A positional index is 2–4 as large as a non-positional index
- Positional index size 35–50% of volume of original text
- Caveat: all of this holds for “English-like” languages
Is there a way we could speed up common/popular phrase queries?

- “Michael Jackson”
- “Britney Spears”
- “New York”

We can store the phrase as another *term* in our dictionary with its own postings list.

This avoids having to do the “merge” operation for these frequent phrases.
Outline for today

- Improvements to basic postings lists
  - Speeding up the merge operation
  - Adding phrase queries and proximity queries

- Text pre-processing
  - tokenizing
  - “all but the kitchen sink” - approaches to token normalization

- Regular expressions in Java (time permitting)
Inverted index construction

Documents to be indexed

Friends, Romans, countrymen.

Text preprocessing

friend, roman, countrymen.

Indexer

Inverted index

friend
roman
countryman

2 4
1 2
13 16
What’s in a document?

- I give you a file I downloaded
- You know it has text in it
- What are the challenges in determining what characters are in the document?

File format:

1. What file types are returned in a Google search?

   There are 13 main file types searched by Google in addition to standard web formatted Microsoft Office formats:

   - Adobe Portable Document Format (pdf)
   - Adobe PostScript (ps)
   - Lotus 1-2-3 (wk1, wk2, wk3, wk4, wk5, wki, wks, wku)
   - Lotus WordPro (lwp)
   - MacWrite (mw)
   - Microsoft Excel (xls)
   - Microsoft PowerPoint (ppt)
   - Microsoft Word (doc)
   - Microsoft Works (wks, wps, wwb)
   - Microsoft Write (wri)
   - Rich Text Format (rtf)
   - Shockwave Flash (swf)
   - Text (ars, txt)

http://www.google.com/help/faq_filetypes.html
What’s in a document?

- I give you a file I downloaded
- You know it has text in it
- What are the challenges in determining what characters are in the document?
  - Language:
    - 莎, Δ, Tübingen, ...
    - Sometimes, a document can contain multiple languages (like this one :)
  - Character set/encoding
    - UTF-8
    - How do we go from the binary to the characters?
  - Decoding
    - zipped/compressed file
    - character entities, e.g. ‘&nbsp;'>
What is a “document”?

- A postings list is a list of documents

  word  ➔  2 ➔ 4 ➔ 8 ➔ 16 ➔ 32 ➔ 64 ➔ 128

- What about:
  - a web page
  - a book
  - a report/article with multiple sections
  - an e-mail
  - an e-mail with attachments
  - a powerpoint file
  - an xml document

- What amount of text is considered a “document” for these lists?
Text pre-processing

- Assume we’ve figured all of this out and we now have a stream of characters that is our document

“Friends, Romans, Countrymen …”

What goes in our dictionary?
A token is a sequence of characters that are grouped together as a semantic unit.
A term is an entry in the dictionary.
Multiple tokens may map to the same term:

- Romans
- roman
- roamns

- roman
Text pre-processing

- Determining the *tokens* and *terms* are the two major pre-processing steps

  "Friends, Romans and Countrymen …"

  \[\text{tokenization} \quad \text{token normalization (determining terms)}\]

  \[\text{Friends, Romans, Countrymen} \quad \text{friend roman countrymen}\]
Outline for today

- Improvements to basic postings lists
  - Speeding up the merge operation
  - Adding phrase queries and proximity queries

- Text pre-processing
  - tokenizing
  - “all but the kitchen sink” - approaches to token normalization

- Regular expressions in Java (time permitting)
Basic tokenization

- If I asked you to break a text into tokens, what might you try?
  - Split tokens on whitespace
  - Split or throw away punctuation characters
Tokenization issues: ‘

Finland’s capital…”
Tokenization issues: ‘

*Finland’s capital*...

What are the benefits/drawbacks?
Tokenization issues: ‘

Aren’t we …

?’
Tokenization issues: ‘

Aren’t we …

Aren’t  Arent

Are n’t  Aren t
Tokenization issues: hyphens

Hewlett-Packard  state-of-the-art

co-education  lower-case

?
Tokenization issues: hyphens

*Hewlett-Packard* *state-of-the-art*

*co-education* *lower-case*

- Keep as is
- merge together
  - HewlettPackard
  - stateoftheart
- Split on hyphen
  - lower case
  - co education

What are the benefits/drawbacks?
More tokenization issues

- Compound nouns: San Francisco, Los Angelos, ...
  - One token or two?
- Numbers
  - Examples
    - Dates: 3/12/91
    - Model numbers: B-52
    - Domain specific numbers: PGP key - 324a3df234cb23e
    - Phone numbers: (800) 234-2333
    - Scientific notation: 1.456 e-10
Tokenization: language issues

Lebensversicherungsgesellschaftsangestellter

‘life insurance company employee’

- Opposite problem we saw with English (San Francisco)
- German compound nouns are not segmented
- German retrieval systems frequently use a compound splitter module
Tokenization: language issues

Where are the words?

- Chinese and Japanese have no spaces between words
  - A word can be made up of one or more characters
  - There is ambiguity about the tokenization, i.e. more than one way to break the characters into words
  - Word segmentation problem

thisissue ?

this issue  this is sue
Outline for today

- Improvements to basic postings lists
  - Speeding up the merge operation
  - Adding phrase queries and proximity queries

- Text pre-processing
  - tokenizing
  - “all but the kitchen sink” - approaches to token normalization

- Regular expressions in Java (time permitting)
Token normalization/Dictionary construction

- We now have the documents as a stream of tokens

  Friends, Romans, Countrymen

- We have two decisions to make:
  - Are we going to keep all of the tokens?
    - punctuation?
    - common words, “to”, “the”, “a”
  - What will be our terms, i.e. our dictionary entries
    - Determine a mapping from tokens to terms
Punctuation characters

- Most search engines do not index most punctuation characters: , . % $ @ ! + - ( ) ^ # ~ ` ' " = ; ? / \ |
Punctuation characters

- Although there are sometimes exceptions...
Stop words

- With a stop list, you exclude from the index/dictionary the most common words

- Pros:
  - They have little semantic content: the, a, and, to, be
  - There are a lot of them: ~30% of postings for top 30 words

- Cons
  - Phrase queries: “King of Denmark”
  - Song titles, etc.: “Let it be”, “To be or not to be”
  - “Relational” queries: “flights to London”
Stop words

- The trend for search engines is to not use stop lists
  - Good compression techniques mean the space for including stopwords in a system is very small
  - Good query optimization techniques mean you pay little at query time for including stop words
Token normalization

- Want to find a many to one mapping from tokens to terms

- Pros
  - smaller dictionary size
  - increased recall (number of documents returned)

- Cons
  - decrease in specificity, e.g. can’t differentiate between plural non-plural
  - exact quotes
  - decrease in precision (match documents that aren’t relevant)
Two approaches to normalization

- Implicitly define equivalence classes of terms by performing operations on tokens
  - deleting periods in a term
  - removing trailing letters (e.g. ‘s’)

- Alternative is to do expansion. Start with a list of terms and expand to possible tokens
  - window → Window, Windows, window, windows
  - Potentially more powerful, but less efficient
Token normalization

- Abbreviations - remove periods
  - I.B.M. → IBM
  - N.S.A. → N.S.A
  - Aug 2005 Google example: C.A.T. → Cat Fanciers website not Caterpillar Inc.

- Numbers
  - Keep (try typing random numbers into a search engine)
  - Remove: can be very useful: think about things like looking up error codes/stacktraces on the web
  - Identify types, like date, IP, ...
  - Flag as a generic “number”
Token normalization

- Dates
  - 11/13/2007
  - 13/11/2007
  - November 13, 2007
  - Nov. 13, 2007
  - Nov 13 ‘07
Token normalization

- Dates
  - 11/13/2007
  - 13/11/2007
  - November 13, 2007
  - Nov. 13, 2007
  - Nov 13 ‘07

![Google search results](image)
Token normalization: lowercasing

- Reduce all letters to lowercase
  - “New policies in …” → “new policies in …”

- Any problems with this?
  - Can change the meaning
    - Sue vs. sue
    - Fed vs. fed
    - SAIL vs. sail
    - CAT vs. cat

- Often best to lower case everything, since users will use lowercase regardless of ‘correct’ capitalization…
Stemming

- Reduce terms to their “roots” before indexing
- The term “stemming” is used since it is accomplished mostly by chopping off part of the suffix of the word

- automate
- automates
- automatic
- automation

- run
- runs
- running

automat
run
Stemming example

Taking a course in information retrieval is more exciting than most courses

Take a course in information retrieval is more exciting than most courses

http://maya.cs.depaul.edu/~classes/ds575/porter.html
or use the class from hw1 to try some examples out
Porter’s algorithm (1980)

- Most common algorithm for stemming English
  - Results suggest it’s at least as good as other stemming options
- Multiple sequential phases of reductions using rules, e.g.
  - sses → ss
  - ies → i
  - ational → ate
  - tional → tion
- http://tartarus.org/~martin/PorterStemmer/
Lemmatization

- Reduce inflectional/variant forms to base form
- Stemming is an *approximation* for lemmatization
- Lemmatization implies doing “proper” reduction to dictionary headword form
- e.g.,
  - *am, are, is* → *be*
  - *car, cars, car's, cars'* → *car*

*the boy's cars are different colors*

*the boy car be different color*
What normalization techniques to use...

- What is the size of the corpus?
  - small corpora often require more normalization
- Depends on the users and the queries
- Query suggestion (i.e. “did you mean”) can often be used instead of normalization
- Most major search engines do little to normalize data except lowercasing and removing punctuation (and not even these always)
Outline for today

- Improvements to basic postings lists
  - Speeding up the merge operation
  - Adding phrase queries and proximity queries

- Text pre-processing
  - tokenizing
  - “all but the kitchen sink” - approaches to token normalization

- Regular expressions in Java (time permitting)
Regular expressions

- Regular expressions are a very powerful tool to do string matching and processing.
- Allows you to do things like:
  - Tell me if a string starts with a lowercase letter, then is followed by 2 numbers and ends with “ing” or “ion”
  - Replace all occurrences of one or more spaces with a single space
  - Split up a string based on whitespace or periods or commas or …
  - Give me all parts of the string where a digit is proceeded by a letter and then the ‘#’ sign
A quick review of regex features

- Literals: we can put any string in regular expression
  - "this is a test".matches("test")
  - "this is a test".matches("hmm")

- Meta-characters
  - \w - word character (a-zA-Z_0-9)
  - \W - non word-character (i.e. everything else)
  - \d - digit (0-9)
  - \s - whitespace character (space, tab, endline, ...)
  - \S - non-whitespace
  - . - matches any character
regex features

- Metacharacters
  - "The year was 1988".matches("19\d\d")
  - "Thereareno spaceshere".matches("\s")
- Java and ‘\’ - annoyingly, need to escape the backslash
  - "The year was 1988".matches("19\\d\\d")
  - "Thereareno spaceshere".matches("\\s")
more regex features

- Character classes
  - `[aeiou]` - matches any vowel
  - `[^aeiou]` - matches anything BUT the vowels
  - `[a-z]` - all lowercase letters
  - `[0-46-9]`
  - "The year was 1988".matches("[12]\d\d\d")

- Special characters
  - `^` matches the beginning of the string
    - "^\d"
    - "^The"
More regex features

- **Special characters**
  - ‘$’ matches the end of the string
    - “Problem 1 - 5 points:”.
      matches(“^Problem \d - \d points$”)
    - “Problem 1 - 8 points”.
      matches(“^Problem \d - \d points$”)
- **Quantifiers**
  - * - zero or more times
  - + - 1 or more times
  - ? - once or not at all
  - “^\d+”
  - “[A-Z][a-z]*”
  - “Runners?”
Regex in java

- java.util.regex.*
  - Patterns
  - Matcher

- For any string:
  - string.matches(regex) - returns true if the string matches the pattern (remember, if it doesn’t have ‘^’ or ‘$’ than it can match part of the string)
  - string.split(regex) - split up the string where the delimiter is all matches of the expression
  - string.replaceAll(regex, replace) - replace all matches of “regex” with “replace”

- LOTS of resources out there!
  - http://java.sun.com/j2se/1.4.2/docs/api/java/util/regex/package-summary.html
Resources for today’s lecture

- IIR 2
- Porter’s stemmer: [http://www.tartarus.org/~martin/PorterStemmer/](http://www.tartarus.org/~martin/PorterStemmer/)
- Skip Lists theory: Pugh (1990)
  - Multilevel skip lists give same $O(\log n)$ efficiency as trees