Text Pre-processing and
Faster Query Processing

David Kauchak
cs458
Fall 2012

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

Administrative

- Tuesday office hours changed:
  - 2-3pm
- Homework 1 due Tuesday
- Assignment 1
  - Due next Friday
  - Can work with a partner
  - Start on it before next week!
- Lunch talk Friday 12:30-1:30

Outline for today

Query optimization: handling queries with more than two terms

Making the merge faster…

Phrase queries

Text pre-processing

Tokenization

Token normalization

Regex (time permitting)

The merge

Walk through the two postings simultaneously

Brutus 2 → 4 → 8 → 16 → 32 → 64 → 128

Caesar 1 → 2 → 3 → 5 → 8 → 13 → 21 → 34

Brutus AND Caesar

Merging

What about an arbitrary Boolean formula?

(Brutus** OR **Caesar)** AND NOT **(Antony** OR **Cleopatra)

- *x* = ***(Brutus** OR **Caesar)***
- *y* = ***(Antony** OR **Cleopatra)***
- *x AND NOT y*

Is there an upper bound on the running time?

- O(total_terms * query_terms)

Query optimization

What about:

Brutus** AND **Calpurnia** AND **Caesar

Brutus** AND (**Calpurnia** AND **Caesar)

(Brutus** AND **Calpurnia)** AND **Caesar

Query optimization

Query: ***Brutus** AND **Calpurnia** AND **Caesar***

Consider a query that is an *AND* of *t* terms.

For each of the terms, get its postings, then *AND* them together

What is the best order for query processing?

Brutus ⇒ 2 → 4 → 8 → 16 → 32 → 64 → 128

Calpurnia ⇒ 1 → 2 → 3 → 5 → 8 → 13 → 21 → 34

Caesar ⇒ 13 → 16

Query optimization example

Heuristic: Process in order of increasing freq:

- *merge the two terms with the shortest postings list*
- *this creates a new AND query with one less term*
- repeat

Brutus ⇒ 2 → 4 → 8 → 16 → 32 → 64 → 128

Calpurnia ⇒ 1 → 2 → 3 → 5 → 8 → 13 → 21 → 34

Caesar ⇒ 13 → 16

Execute the query as (*Caesar AND Brutus) AND Calpurnia*.

Query optimization

Query: ***Brutus*** *OR* ***Calpurnia*** *OR* ***Caesar***

Consider a query that is an *OR* of *t* terms.

What is the best order for query processing?

Same: still want to merge the shortest postings lists first

Brutus	➡	2 → 4 → 8 → 16 → 32 → 64 → 128
Calpurnia	➡	1 → 2 → 3 → 5 → 8 → 13 → 21 → 34
Caesar	➡	13 → 16

Query optimization in general

(madding OR crowd) AND (ignoble OR NOT strife)

Need to evaluate OR statements first

Which OR should we do first?

- Estimate the size of each OR by the sum of the posting list lengths
- NOT is just the number of documents minus the length
- Then, it looks like an AND query:
  - *x* AND *y*

Outline for today

Query optimization: handling queries with more than two terms

Making the merge faster…

Phrase queries

Text pre-processing

Tokenization

Token normalization

Regex (time permitting)

The merge

Walk through the two lists simultaneously

word1 2 → 4 → 8 → 16 → 32 → 64 → 128

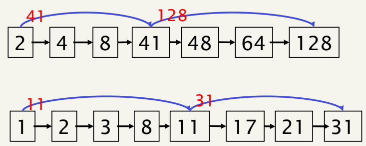
word2 1 → 200

O(length1 + length2)

Can we make it any faster?
Can we augment the data structure?

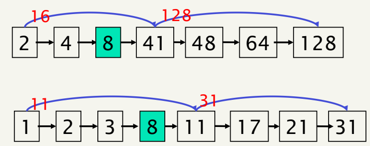
12

Augment postings with skip pointers (at indexing time)

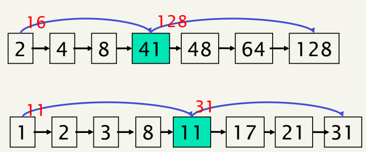


How does this help?

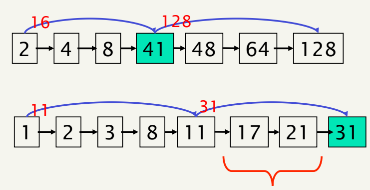
Query processing with skip pointers



Query processing with skip pointers



Query processing with skip pointers

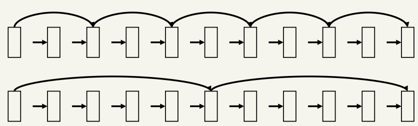


we skip these entries

Where do we place skips?

Tradeoff:

- More skips → shorter skip spans ⇒ more likely to skip. But lots of comparisons to skip pointers. More storage required.
- Fewer skips → few pointer comparison, but then long skip spans ⇒ 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

Query optimization: handling queries with more than two terms

Making the merge faster…

Phrase queries

Text pre-processing

Tokenization

Token normalization

Regex (time permitting)

Phrase queries

We want to be able to answer queries such as "***middlebury college***"

"I went to a college in middlebury" 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

word1 2 → 4 → 8 → 16

word1 2: ⟨3,16,20⟩ → 4: ⟨39⟩ → 8: ⟨4, 18, 40⟩ → 16: ⟨7⟩

docID: ⟨position1,position2,…⟩

Positional index example

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

to:
1: ⟨4,17,32, 90⟩
2: ⟨5, 50⟩
4: ⟨12,13,429,433,500⟩
5: ⟨4,15,24,38,366⟩

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 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⟩

to:
4: ⟨12,13,429,433,500⟩

→

be:
4: ⟨17,191,291,430,434⟩

to:
4: ⟨13,14,430,434,501⟩

Processing a phrase query: "to be"

Find all documents that have 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⟩

to:
4: ⟨12,13,429,433,500⟩

→

be:
4: ⟨17,191,291,430,434⟩

to:
4: ⟨13,14,430,434,501⟩

What about proximity queries?

Find "middlebury" 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

Is this efficient?

No, Naïve algorithm is inefficient, but doing it efficiently is a bit tricky

Positional index size

How does positional indices affect the posting list size?

Makes it significantly larger!

Rather than only keeping track of whether or word occurs or not, have all occurrences of a word

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%

Document size	Postings	Positional postings
1000	?	
100,000		

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%

Document size	Postings	Positional postings
1000	1	
100,000	?	

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%

Document size	Postings	Positional postings
1000	1	?
100,000	1	

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%

Document size	Postings	Positional postings
1000	1	1-2
100,000	1	?

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%

Document size	Postings	Positional postings
1000	1	1-2
100,000	1	100

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

Popular phrases

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 it's own postings list

This avoids having do do the "merge" operation for these frequent phrases

Outline for today

Query optimization: handling queries with more than two terms

Making the merge faster…

Phrase queries

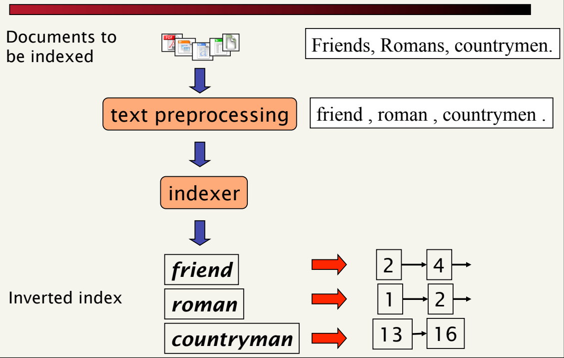
Text pre-processing

Tokenization

Token normalization

Regex (time permitting)

Inverted index construction

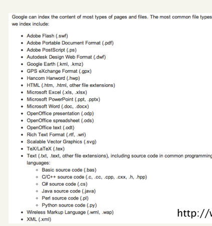


What's in a document?

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:


Google can index the content of most types of pages and files. The most common file types we index include:

- Adobe Flash (.swf)
- Adobe Portable Document Format (.pdf)
- Adobe PostScript (.ps)
- Autodesk Design Web Format (.dwf)
- Google Earth (.kml, .kmz)
- GPS eXchange Format (.gpx)
- Hancom Hanword (.hwp)
- HTML (.htm, .html, other file extensions)
- Microsoft Excel (.xls, .xlsx)
- Microsoft PowerPoint (.ppt, .pptx)
- Microsoft Word (.doc, .docx)
- OpenOffice presentation (.odp)
- OpenOffice spreadsheet (.ods)
- OpenOffice text (.odt)
- Rich Text Format (.rtf, .wri)
- Scalable Vector Graphics (.svg)
- TeX/LaTeX (.tex)
- Text (.txt, .text, other file extensions), including source code in common programming languages:
  - Basic source code (.bas)
  - C/C++ source code (.c, .cc, .cpp, .cxx, .h, .hpp)
  - C# source code (.cs)
  - Java source code (.java)
  - Perl source code (.pl)
  - Python source code (.py)
- Wireless Markup Language (.wml, .wap)
- XML (.xml)

http://www.google.com/help/faq_filetypes.html

What's in a 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. ' '

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*** …"

Brutus → 2 → 4 → 8 → 16 → 32 → 64 → 128

Calpurnia → 1 → 2 → 3 → 5 → 8 → 13 → 21 → 34

Caesar → 13 → 16

Dictionary

What goes in our dictionary?

Text pre-processing

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:

token	term
Romans	roman
roman	roman
roamns	roman

Text pre-processing

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

"***Friends, Romans and Countrymen*** …"

↓

tokenization → Friends , Romans , Countrymen

↓

token normalization (determining terms) → friend roman countrymen

Outline for today

Query optimization: handling queries with more than two terms

Making the merge faster…

Phrase queries

Text pre-processing

Tokenization

Token normalization

Regex (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*...

Finland

Finland ‘ s

Finland ‘s

Finlands

Finland s

Finland’ s

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

What are the benefits/drawbacks?

Split on hyphen
- lower case
- co education

More tokenization issues

Compound nouns: San Francisco, Los Angeles, …
- 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

莎拉波娃现在居住在美国东南部的佛罗里达。

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

Query optimization: handling queries with more than two terms

Making the merge faster…

Phrase queries

Text pre-processing

Tokenization

Token normalization

Regex (time permitting)

Token normalization/ Dictionary construction

We now have the documents as a stream of tokens

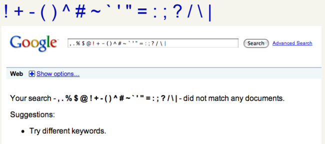
Friends , Romans , Countrymen

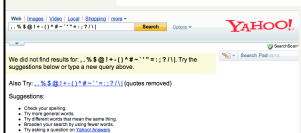
We have two decisions to make:

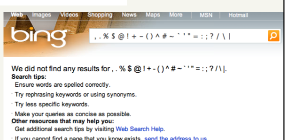
- Are we going to keep all of the tokens?
  - punctuation?
  - common words, “to”, “the”, “a”
- What will be our *terms*, i.e. our dictionary entries
  - Determine a mapping from *tokens* to *terms*

Punctuation characters

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

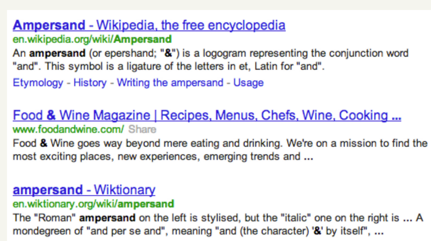






Punctuation characters

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

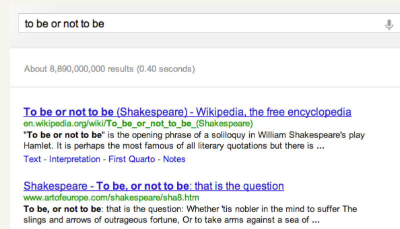
Cons

- 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 stop words 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’)

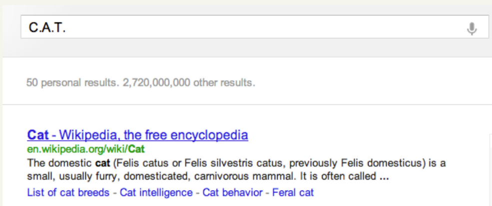
Alternative is to do expansion. Start with a list of terms and expand to possible tokens

- window → Window, Windows, window, windows
- Potentially more powerful, but less efficient

Token normalization

Abbreviations - remove periods

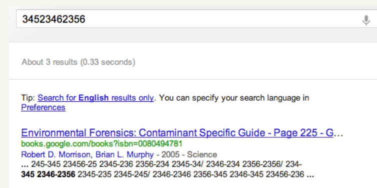
- I.B.M. → IBM
- N.S.A. → N.S.A
- Google example: C.A.T. → Cat *not* Caterpiller Inc.



Token normalization

Numbers

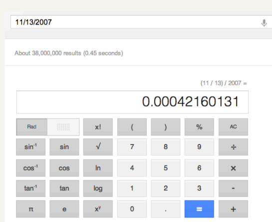
- Keep (try typing random numbers into a search engine)
- Remove: can be very useful: think about things like looking up error codes/stack-traces on the web
- Identify types, like date, IP, …
- Flag as a generic “number”



Token normalization

Dates

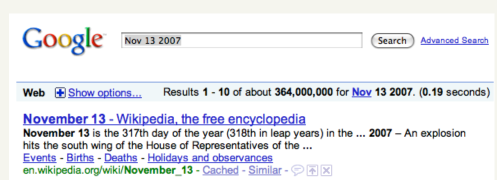
- 11/13/2007
- 13/11/2007
- November 13, 2007
- Nov. 13, 2007
- Nov 13 ‘07



Token normalization

Dates

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



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 → *automat*

run
runs
running → *run*

Stemming example

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

Take a cours in inform retriev is more excit than most cours

http://maya.cs.depaul.edu/~classes/ds575/porter.html
or use the class from assign1 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

Query optimization: handling queries with more than two terms

Making the merge faster…

Phrase queries

Text pre-processing

Tokenization

Token normalization

Regex (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")
- "Therearenospaceshere".matches("\s")

Java and '\' - annoyingly, need to escape the backslash

- "The year was 1988".matches("19\\d\\d")
- "Therearenospaceshere".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/docs/books/tutorial/essential/regex/intro.html
  - http://java.sun.com/j2se/1.4.2/docs/api/java/util/regex/package-summary.html