Search Engines

WS 2009 / 2010

Lecture 1, Thursday October 22nd, 2009

Prof. Dr. Hannah Bast
Chair of Algorithms and Data Structures
Department of Computer Science
University of Freiburg

Overview of this Lecture

- Introduction
  - a bit about myself
  - the kind of work we do in our group
  - teaching style, project after the course ends
- Search
  - parsing
  - building an inverted index
  - querying an inverted index
  - a simple space and time analysis
- Exercises
  - go over Exercise Sheet 1, explain course Wiki

About myself

- Education
  - Ph.D. at Saarland University, 1999
  - researcher (W2) at the MPI for Informatics, Saarbrücken
  - researcher (W2) at MMCI Cluster of Excellence, Saarbrücken
  - professor (W3) in Freiburg since September 2009
- Real work
  - worked at Siemens a long time ago
  - consulted for many (search engine) companies
  - worked at Google Zürich for the last 1 ½ years
- Research interests
  - I do and like what I call Applied Algorithmics

CompleteSearch Demo

- Developed by our group since 2005 public demos
- Show + explain the following
  - smart + complex searches, but still very fast comparison
  - show variety of collections / applications
  - user interface, show JavaScript source
  - TCP traffic, show via FireBug / CS Infobox
  - web server (Apache), show access log
  - middleware code (PHP), show access log
  - backend, show server log for DBLP
  - CompleteSearch code, and the algorithms behind

You will learn about all of this in this lecture !

Web Search vs. Domain-Specific Search

- Web Search
  - ranking is extremely important
  - recall is not an issue for popular queries and hopeless for many expert queries
  - Spam, spam, spam, spam, spam, spam, spam, spam, …
  - very limited resources for fancy stuff
- Domain-Specific Search
  - recall is important example
  - Spam is not an issue
  - more resources to do fancy stuff (still has to be fast though)

Google is great on Web Search, we do Domain-Specific Search

Searching by Scanning (grep)

- That's what a Unix / Linux grep does
- It's not so bad, a modern computer can …
  - … scan 100 MB from disk per second
  - … scan 1 GB of memory per second
- However grep is line-based
  - finds lines that match a given pattern
  - but there are extensions which do Google-like search, for example, agrep

Parsing / Tokenization

- Conceptually simple:
  - just break a given text into words / tokens
- But:
  - 高見 順 : 娘よりの聞書きにつき誤引用の可能性あり
  - ich schwÃ¶re bei^M meiner MÃ¶hre ...
  - Donaudampfschifffahrtsgesellschaft
  - stemming: for example, search eggs, find egg

for the exercises you can do something simple

The Inverted Index

- Idea
  - like the index in the back of a book
  - but for *every* word that occurs
- Specifically
  - for every word in the collection, a list of the ids of the documents containing it (called inverted list)

    informatik: Doc12, Doc57, Doc59, Doc61, Doc77, …
- Construction
  - it's basically one big sort: parsing outputs the word occurrences sorted by document and position, for the inverted index we need it sorted by word **show example**

Index Construction = Sorting

Doc1

What a stupid document.

Doc2

Another stupid document.

parser

what	1
a	1
stupid	1
document	1
another	2
stupid	2
document	2

sorting →

a	1
another	2
document	1
document	2
stupid	1
stupid	2
what	1

Alternatively, use Hashing

- Have a hash map words → list of doc ids
  - in C++: hash_map<string, vector<int> >
  - whenever you encounter a word for the first time, insert it into a hash map with an empty list
  - append subsequent occurrences to that list
- Complexity, where N = total number of word occurrences
  - Sorting takes time $O(N \cdot \log N)$
  - Hashing takes constant time per word, hence $O(n)$
  - Still it's not so clear which approach is better, why?
    - each hash operation is likely to be a cache miss
    - hashing only works when the index fits in memory
    - more about this in one of the next lectures

Space Analysis

- Total size of the inverted index?
  - one inverted list entry per word occurrence
  - but we have an id instead of a full word
  - that already gives some kind of compression
  - later in the course we will compress even more
  - size of an index = 10 – 20% of whole collection

Querying an inverted index

- Example query: informatik freiburg
  - fetch the two inverted lists
  - intersect them

  informatik: Doc 12, Doc14, Doc27, Doc54, Doc 55, …

  freiburg: Doc 5, Doc 12, Doc 13, Doc14, Doc67, …

  intersection → Doc 12, Doc14, …
- Efficiency
  - important that the lists are sorted by doc id
  - then cost of intersection = O(log k · sum of list sizes)
  - why the log k?

Intersection of multiple lists

- Assume we have three lists

informatik:	Doc 12, Doc14, Doc27, Doc54, Doc 55, …
freiburg:	Doc 5, Doc 12, Doc 13, Doc14, Doc67, …
master:	Doc 7, Doc 12, Doc14, Doc 38, Doc 72, …

- Algorithm:
  - for each list maintain the current position in the list, and the doc id at that position in a priority queue
  - at each step, find those of the current positions with the smallest entry, and advance that position **show with lists above**
  - with a priority queue this operation takes log k time, where k is the number of items in the queue (here: the number of lists)
  - Note: a trivial implementation of a priority queue (always scan all items to find the smallest element) would take time k

How long are the inverted lists

- Zipf's law:
  - The i-th most frequent word in the collection occurs approximately $\varepsilon \cdot N \cdot 1 / i$ times, for some constant ε and N = total num of word occurrences
  - Exercise: verify this for your collection. What is your ε ?
- So with k query words with ranks $r_1, \ldots, r_k$:
  - the total length of the lists is $\varepsilon \cdot N \cdot \Sigma\, 1 / r_i$
  - let's compute how much this is in expectation …