Information Retrieval: Assignment 1

Sawood Alam
salam@cs.odu.edu

September 27, 2011

Abstract

This document contains solutions and discussions on exercise questions 1.4, 2.3, 3.6, 3.7 and 3.8 from the textbook, Search Engines Information Retrieval in Practice - W. Bruce Croft, Donald Metzler and Trevor Strohman.

1 Problem 1.4

1.1 GitHub (http://github.com/)

It is a Git based code hosting service.

- It gives option of fielded searching (repositories, users, code and language) in the public areas. Due to the fact that it has no categorized browsing feature, search becomes the essential and most obvious method of using this service (unless the user has direct URLs or has subscribed certain projects).
- Searching is most probably based on search index. It is a large scale web service and search on such a scale is infeasible using grep style searching.
- It is hard to claim whether they are using any ranking technique or not. But they have many attributes available for ranking (e.g. repository size, number of forks, number of watchers, last activity time and number of downloads etc.).

1.2 SourceForge (http://sourceforge.net/)

It is another open source code hosting service.

- Searching is one of the main ways to interact with the service. Although, they have well structured, multi level categorization to facilitate browsing. There seems no advanced or fielded searching interface but search results have option for filtering further on various attributes.
- It is another web service of huge scale. They definitely have search indexes to facilitate searching on such a scale.
- They do have ranking because, they have an option of sorting by relevance (but mechanism or algorithm is unknown).

1.3 XenForo (http://xenforo.com/community/)

It is a commercial forum (discussion board) software written in PHP and using MySQL as database engine.

- Although, discussion boards have nested categories and sub-categories to facilitate browsing the topic of interest. But in the end, final listing of individual items in any (sub-)category (or latest posts, recent activity) is performed using database driven search only. Also, fielded and advanced searching in the entire forum data along with options to filter and sort results is present in this software.
- I happen to have access to their code and database schema. Based on that I know that they are using full-text search feature of MySQL. This full-text search is based upon creating indexes of type FULLTEXT on searchable fields in MySQL. But they have an option for custom search solutions (like Solr) as well.
- They use no specific ranking in their default setup. But, if a custom search solution is integrated then this might not be true.

1.4 WordPress (http://wordpress.com/)

WordPress is an open source blogging engine written in PHP and using MySQL as database.

- Searching in blogs is used occasionally. It is usually published by individuals on indefinite intervals and being read by readers usually with the help of feed aggregators etc. Because of this nature of the blogs regular readers prefer to access it in canonical order. Also, discovery of the previous content (and new traffic) happens usually with the help of web search engines. But still it has sophisticated database driven search solution that can work nicely on the scale of usual blog. For large scale blogs, custom solutions and/or plugins can be used.
- By default, WordPress do not use MySQL full-text search capabilities. Mainly because of the reason that full-text indexing support was added in later versions of MySQL. To maintain the compatibility with old

MySQL databases, it is not included in the default setup yet. But one can easily tweak it to take advantage if suitable resources are available.

1.5 IndexTank (http://indextank.com/)

It is a restful web service to facilitate hosted indexing and searching.

- Content owners can index their content using indexing API and then using searching API the search can be performed on the indexed objects. It supports fielded, full-text and boolean searching options.
- It is not using grep style searching. As the name suggests it uses indexes to facilitate searching.
- Ranking is very customizable and tunable at the time of indexing or later on.
- I have used this service to facilitate searching on final project of "Introduction to Digital Libraries" class. I was very satisfied with its indexing, ranking, searching options as well as full Unicode support.

1.6 Discussion

- I was looking for some web services that are using file based (grep style) searching but it turns out to be hard to spot one. There are some NoSQL applications using MongoDB or similar document based schema-less storage. But, according to the documentation of InnoDB it supports indexing as well (http://www.mongodb.org/display/DOCS/Indexes) to improve the query performance and scale.
- Searching in grep style has some advantages like, possibility of using RegEx, getting fresh search result (real-time) and it is fast on small text data. But it is really slow on large scale and it is hard to implement fielded, advanced and boolean searching. Also, ranking the search result is a challenge. Indexing might require lots of disc space as index footprint and extra CPU cycles to create and update indexes but improves the searching time. There is clearly a trade off between

time and space. Added to that, grep is not a feasible solution on large scale.

- One of the examples I gave was IndexTank, a hosted searching solution. one might argue that why not to use Google site search (or other similar site search solutions) as ODU uses it. The problem in this case is the lack of control over the indexable fields and ranking mechanism. Such site search solutions that work on the basis of crawling the web and indexing by themselves, can only index based upon the publicly visible objects and properties. While using services like IndexTank can provide option to index and rank publicly hidden objects (only shown to individuals having access permission) and hidden properties like last modified date or other fields (assuming that it is not exposed on user interface).

2 Problem 2.3

A simple Filter Engine (Figure 1) might have following components.

- Feed stack (periodically renewing),
- Feed processor,
- Query store,
- User store (associated with query store) and
- Mailer stack

2.1 Workflow

Feeds are fetched from various sources periodically and stored in feed stack. Then feed processor takes one feed at a time and runs all the queries against that feed one after the other. If query is successful, all the users associated with that query are pushed in mailer stack along with the feed. Once all the queries are processed against certain feed, mailer stack is ready to deliver emails related to that feed. Mailer stack should avoid having duplicate users

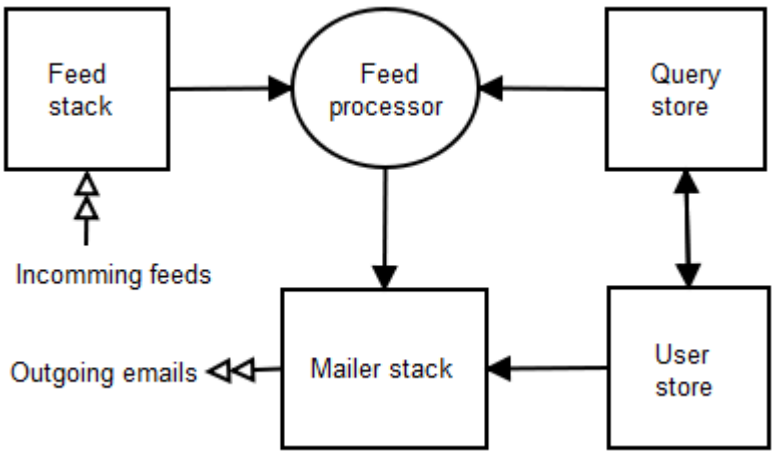

Figure 1: Filter Engine

from different queries for the same feed otherwise, same email will be sent to certain subscribers multiple times.

This is the simplest model which can work for small set of subscribers, feeds and queries but, this simple model cannot scale. Such a system requires consistent and quick delivery of the content to the subscribers. Almost instant delivery might be important in some cases where, delayed delivery might make the document useless for the user. Slow feed processor, massive number of queries in the query store (due to large number of subscribers) and too many feed sources are some of the factors that might cause feed stack overflow. Also, an inefficient mailer system might result in mailer stack overflow.

On large scale, distributed processing will be required. A bad architecture might result in a tightly integrated system that is hard distribute. lack of independence will cause unnecessary communication overhead among distributed processors.

2.2 Optimization for scale

2.2.1 Clustering queries/profiles

Clustering queries in an N dimensional vector space and then locating feed in that space might help in quickly identifying the queries yielding success against that feed. This way, feed processing will be done against a smaller set of queries and lots of queries will be avoided, considering them irrelevant to the document. This query clustering may also help in distributed architecture. Closely related queries can be placed in one query store close to a dedicated feed processor for similar queries and minimize the communication overhead. If maximum coverage is crucial then another process with long interval (like overnight) can run for the whole feed stack against whole query store (minus those, already processed) on the cost of delayed delivery to cover the outliers.

2.2.2 Avoiding duplicate emails

One subscriber may have multiple queries associated in the query store and more than one queries might result success against the same feed. This might result in sending duplicate emails to the same user for the same feed. There might be few ways to avoid this situation. If the feed is the key and a list of qualified emails is the value (other way round is also possible but more memory consuming due to duplicate values for several keys) then, simplest way is to sort the list of emails and make the list unique before sending out emails. But, sorting the list and making it unique might be costly process if the number of recipients is large. Using a SET data structure may also avoid the duplicate by the virtue of set property. Another efficient way could be to avoid insertion of duplicate emails in the mailer stack in the first place by making copy of all users in memory and instead of copying them in the mailer stack, just move them. But, large user-base might not fit in the memory hence, another way to deal with this could be to keep a flag bit in the user store. Set that flag once a user is queued in the mailer stack and reset all flags after each feed processing. If multiple feed processors are using the same user store then multiple such flags can be placed for every processor.

2.2.3 Batching emails

The system might be configured to send emails in batches by bundling several qualifying documents in a single email to the user. To implement this scenario, emails might be used as key in the mailer stack and a list of qualifying feeds can be pushed as value for individual emails. Depending upon the delivery scheduling policy, mailer then decides to send out emails. In this scenario, keys (emails) having same vale (same list of feeds) can be batched together to exploit the optimisation of email servers against (mailing list style) batched emails.

Considering limited scope of this discussion I am not going to explore further possibilities, issues and communication overheads associated with above optimizations when, this is implemented on a distributed architecture.

3 Problem 3.6

Programming the automation of web forms is not a challenge. Using modern headless browsing libraries like Mechanize for Ruby or simply parsing the form and extracting names of the fields then preparing a GET request (or POST) is pretty easy. The real challenges are to identify the non-destructive forms and finding values suitable for free text entry elements. I will discuss these two challenges separately. (Discovery of DOM generated using JavaScript/AJAX is out of scope of this discussion.)

3.1 Identification of non-destructive forms

If the web was written on ideal measures/standards then this was not a challenge at all. According to REST methodology, any destructive form should either have POST or PUT method. While, forms used for discovering data should have GET method set. But, if we go by this assumption, we will end up missing several important non-destructive forms and vice versa, because of several non standard implementations.

To handle this situation, we might use heuristic approaches to identify potential non-destructive or destructive forms. This might also require need of

training an AI model to help classifying the forms. Some of the attributes that might be helpful in identifying form class are as follows.

- HTTP method - GET/POST (PUT is not supported by browsers hence, rare to be scene in web forms)
- Value of form action attribute - Verbs used in action attribute might be helpful (post-comment.php, update-profile.aspx vs. fetch-users.php, searchSongs.do), while some might be vague like "submit.php".
- Title of the submit button - Verbs used as label of form submit button might also be good indicator ("Post Comment", "Submit Changes" vs. "Find", "Lookup"), while some might be vague like "Go".
- CSS class and id attributes might also be good indicators if their values have some semantic meaning.
- Placement/position of form in the document structure may also be indicative of its type.
- Existence of HTML5 search input type. (strong indicator of non-destructive form)
- Existence of textarea element or rich text editor. (strong indicator of destructive form)
- Existence of password field. (usually useless for deep web discovery)
- Some standard or popular CMSs like WordPress, Joomla, MediaWiki, vBulletin and phpBB etc. have meta tags that tells about the generator system. Profiling based upon that may also help a lot because of their well known properties and style.

3.2 Identifying suitable values

If the form element type is option/select, radio button, checkbox, hidden, then possible values can be extracted from the form itself. And all possible combinations will ensure enough coverage. But if the element type is free text then it is a challenge to find suitable values to fill. several tries might lead to false input and good coverage will be hard to achieve. There might be some indicators in or around the element to help identify the set of values

(e.g. full names, states, cities, countries, number ranges, dates etc.) to try. Here are some of the possible indicators.

- Label of the field. (e.g. “Full Name”, “Search for”)
- Name attribute of the field. (e.g. “current-city”, “q”)
- Leading or following text node. (sometimes standard label elements are not used.)

This mechanism requires some standard sets to do exhaustive hit and trial attempts. It may result in very poor throughput.

4 Problem 3.7

4.1 Code

```
#!/usr/bin/ruby

# Top-down directory structure traversal module
require "find"

# Constant configs
WEBROOT = "/home/salam/public_html"
ROOTURL = "http://www.example.com"

# Template for each <url /> block of sitemap
TPL = <<EOS
  <url>
    <loc>URI</loc>
    <lastmod>LMDATE</lastmod>
    <changefreq>monthly</changefreq>
    <priority>0.8</priority>
  </url>
EOS
```

```
# String holding content of XML. This will grow iteratively
sitemap = <<EOS
<?xml version="1.0" encoding="UTF-8"?>
<urlset xmlns="http://www.sitemaps.org/schemas/sitemap/0.9">
EOS

# Iteration under Webroot directory.
Find.find(WEBROOT) do |p|
  if File.directory?(p)
    # If directory is not readable/executable by others
    unless File.world_readable?(p)
      # Do not look any further into this directory.
      Find.prune
    end
  else
    # Include file in the sitemap only if it is readable by others
    if File.world_readable?(p)
      # Extract file modification time in W3C date-time format
      lmd = File.new(p).mtime
      # Substitute WEBROOT path with ROOTURL
      uri = p.to_s.sub(WEBROOT, ROOTURL)
      # Substitute suitable values in <url /> block template
      # and append it to existing sitemap string
      sitemap += TPL.gsub('URI', uri).gsub('LMDATE', lmd)
    end
  end
end

# Append closing </urlset> tag in the sitemap string
sitemap += "</urlset>"

# Write built sitemap string in appropriate sitemap file
rem = File.new("#{WEBROOT}/sitemap.xml", "w")
rem.puts sitemap
rem.close
```

4.2 Discussion

Code is self explanatory with suitable inline comments. This is a Ruby script, hence more readable but execution time my not be as good as pure C implementation. I have used an in-built module called 'find' to avoid writing recursive directory traversal code myself.

I have built whole sitemap in the memory and at the end it was written on the disk to avoid multiple file writes that will affect the performance significantly. Usually, sitemaps are of limited size (few MBs), hence it is very practical to build them in the memory.

According to Google's Webmaster sitemap best practice guidelines it should not contain more than 100 links. If this guideline is to be considered, few simple edits in the code will do the trick, without changing main algorithm. Essentially it will require a url block counter and a sitemap file counter. After reaching the url count threshold, file writing module should be called that will write the content of "sitemap" string in a sitemap file (file number will be appended to file name) and reset sitemap string to its initial state, increment the sitemap file counter, reset url counter and return the execution pointer to the parent procedure. In addition to that, a master sitemap file may be created to point to all sitemap files (or entries in the robots.txt will be enough). Too keep the code clean, I did not include this logic in there.

While constructing the sitemap, I used plain string instead of creating and XML Node object for the simple reason of performance and memory usage. A string serialization will take significantly less memory and manipulation time (in this case) as compared to the equivalent XML Node object.

URL escaping is not required because of the fact that UNIX file system based file paths will never have a URL incompatible character in them. Hence, adding code to escape the URL will only increase the execution time overhead.

UNIX file Last-Modified time was taken and converted to W3C standard date-time format.

Apache restricts access to any file (irrespective of its permissions) under a non-executable/non-accessible directory at any depth. E.g. "/path/to/a/public/file.txt" will not be web accessible if any of the directories in the entire path does not

have read/execute access. This measure was taken into consideration while building the sitemap.

No file is included in the sitemap unless it is readable by others/world.

5 Problem 3.8

Answer to this question may vary "a little", depending upon the implementation of the distributed crawler. But in general case, having different seed URLs will not improve the performance or coverage on large scale web.

If the number of seed URLs is considerably large, distribution of hash function (to decide the target crawler of a URL) is uniform and both crawlers use same hashing function (i.e. no central server) then we can assume that in the very beginning of URL exchange between the two crawler processes, both will have half of the total unique seed URLs. This means, having same seed URLs on both the crawlers will effectively make the number of seed URLs half as compared with having different seed URLs for both the crawlers in the first place.

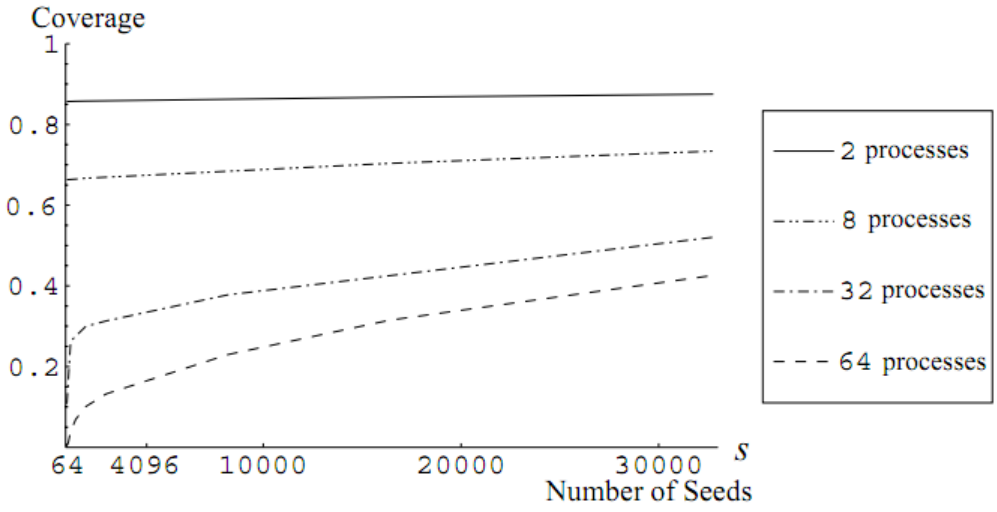

Figure 2: Number of seed URLs vs. Coverage (Source: Parallel Crawler [1])

Since, no crawler is going to download documents belonging to the other

crawler, hence the only metric to be considered is the number of seed URLs. The paper, "Parallel Crawlers"[1], has a graph of Number of seed URLs vs. Coverage (Figure 2). The graph clearly shows that for just two processes, number of seed URLs make no significant difference in coverage. But, for higher number of processes, having small seeds have some effects on coverage.

References

[1] J. Cho and H. Garcia-Molina. Parallel crawlers. In *Proceedings of the 11th international conference on World Wide Web*, pages 124–135. ACM, 2002.