CONTENTS

INTRODUCTION

Search engines are essential for every internet user. They are in charge of referencing an ever-growing amount of web data and matching the referenced documents with user queries. It could be said that they form our perception of the web. This cornerstone role is only endorsed by few companies, using complex mechanisms about which little is known. The main objective of this research is to devise methods that can reveal some of these internal mechanisms. More precisely, we target the design of adversarial transparency approaches to search engines.

It is difficult not to agree that search engines referencing impact on a society on the web. Internet user can find only the information that is considered as relevant by the search engine. Furthermore, the number of website visits highly depends on its positioning and ranking. These underlying crawling and ranking algorithms are mostly opaque or only revealed by the goodwill of the search engine companies. Therefore, we audit the latter as online black boxes to analyse what are the search engines doings. As the first objective, we set up a monitoring system to interface with a particular search engine. The second goal is to demonstrate the application of our approach with two use-cases – analysing collected data and robots' records and analysing query rankings.

The matching monitoring system is designed as a Petri dish. In short, a Petri dish is a container that biologists use in different scientific tests. One of them is to study the behaviour of ecosystem in response to adding here some outer organism. Imagine that each search engine has its own medium consisting of search robots, referenced web pages, ranking algorithms and other tools and data. In our research, we need to attract search robots to explore the search engine's "ecosystem". Therefore, we use a website, which serves as a honeypot for robots, as an "outer organism". We observe actions in the engine's "dish", that is why we call this structure a monitoring system.

Setting up a Petri dish implies analysing the collected records. While crawlers visit our honeypot, we gather information they produce. After that, we process the data in order to find engine's behaviour patterns and characteristics.

Another application of set up system is analysing query rankings. Here we use attractive honeypot to post the set of simple documents. Firstly, we chose some known implemented ranking functions and query them with the target keywords locally to get corresponding orderings of the prepared documents. Secondly, we feed search engines with the aforementioned queries to collect the orderings produced by engines' rankings. Finally, we compare the orderings obtained locally and remotely to measure the similarity of known ranking functions and unknown ones which are used by the researched search engines.

What makes this work novel is studying search engines from the adversarial perception. We act like search engines never collaborate with us and never let us know they collect the data. In more detail, we do not use the public knowledge like some engines share information among them, engines usually announce their search bots and so on. Moreover, in fine, we reveal information that is already (mostly) disclosed. This allows to confirm the veracity of the extracted data, and proves the capacity of our approaches to reveal other information even without search engine's consent.

The main cause of studying search engines is their relevance itself. They have a huge influence on the web and on internet users particularly. Our research is experimental and introductory, but the approach we use may lead to further investigations in this area. In addition, the results of ranking analysing might be useful in Search Engine Optimization sphere.

There are four chapters in this research. The first one stands for introduction to the subject area. There are essential terms from the Search Engines sphere and related fields, also we clarify the goals for the work here. The second chapter delineates monitoring system implementation and its use-cases. The third chapter con-

tains query ranking analysing. And the fourth chapter is an additional one, studying the robots' refreshing of dynamic and static pages.

CHAPTER 1. DESCRIPTION OF THE SUBJECT AREA AND INTRODUCTION TO THE SPECIFICITY OF WORK.

We are going to introduce an essential terminology from the search engines sphere, outline honeypot website and chosen ranking functions, delineate researched search engines in this chapter. We will also specify the goals for the work and the research questions.

1.1. An essential terminology from the Search Engines sphere.

A **search engine** is a software system which is developed to search the World Wide Web (WWW) systematically for certain information specified in a search query. The search results are usually presented as a list of web pages called **SERPs** (Search Engine Result Pages). These pages are generally a list of links to other web pages, images, videos, research documents and other file types. In our research, we focus on textual data only.

Each search engine usually relies on a web crawler. A **web crawler**, which is also referred to as **search robot**, **search bot**, **spider** or **crawler**, is a software program that automatically explores the WWW by recursively crawling hypertext links on each page it encounters.

Crawlers used by search engines are run for the purpose of updating a search engine's web content. It could be either an adding new web pages or an amendment to the existing ones. This process of storing and organizing data, gathered during the crawling, is called **indexing**. Sometimes the amendment is also named **re-indexing**. When a page is in the engine's index, it is ready to appear on SERP as a result of relevant queries. An **index** can be also interpreted as a database of all the content that search engines have discovered and admitted as relevant information.

One more essential term we refer to in our work is a **SEO** (Search Engine Optimization). Basically, SEO is "the process of making your site better for search engines" [10]. It implies a set of tips to make a site indexed and re-indexed quickly, and a range of techniques to improve a site's SERP position. Some of them are

also provided by Google in its Search Central Documentation [10]. It is no surprise people care about SEO since the number of visitors depends crucially on website's position on SERP. For example, according to [21] about the thirty percent of all organic clicks go to the first link on Google SERP, and the rate decreases steeply with the link's position increasing.

A considerable aspect of SEO is to pay attention to website's keywords. **Keywords** are words and phrases that define what a site's content is about. Also, they are the text of user's search query. Keywords are significant because they are the connecting link between what people search for and the information a web page provides to fill that necessity. For the better SEO results, keywords should be both included in a web page's title and substance so that both search engines and people see what content is about.

When a user enters search terms, an engine evaluates the importance of a webpage with respect to the search terms, which is called relevance. Thus, an engine's search results are ordered by the relevance to a search query. This ordering is called **ranking**. Algorithms behind ranking constantly change for a purpose of improving the quality of search results.

To automate the querying of search engines, and the recording of the collected SERP, we rely on techniques called **web scraping**. It is a process of harvesting the data from a webpage's source code.

1.2. Description of the chosen search engines.

We focus our study on ten dissimilar search engines. Supporting each requires some specific code, and since the market is very concentrated those ten already cover a significant share of the world's search engines. They are described in the appendix A.

1.3. Honeypot website outline.

Originally, honeypot is a mechanism for hackers' and spam's detection. It is usually a monitored part of a website, containing information to attract attackers.

In our work, the honeypot is a website to attract robots, that is why we need to optimize it for search engines. Moreover, the site has to be easily found on the web. Consequently, honeypot requirements are the following:

— looking like a normal web page, some popular topic that search engines would like to serve
— light enough to be loaded quickly by search robots
— containing a unique word so that we are able to extract the record corresponding to our website easily among all the records of search engines

1.4. Description of the chosen ranking functions.

Relevance of an engine's search result is a subjective measure, and every definition of the relevance generates a ranking function. Therefore, the set of possible ranking functions is uncountable. In our research, we focus on three state-of-the-art functions from the BM25 family. In the BM25 model the documents are presented as multisets of their words where the word's frequency prevails over words order and grammar. Therefore, the documents are ranked by "the query terms appearing in each document, regardless of their proximity within the document" [14]. The first function from this family is **BM25Okapi**. It is usually referred to as a baseline function. The **BM25L** is an improved version of the base BM25, which fixes unfairly preferring shorter documents to longer ones during the document length normalization [1]. The **BM25Plus** also solves the problem of shorter documents, but in a more common way – it adds lower-bounding of "the contribution of a single term occurrence" [1]. In other words, BM25Okapi ranks the documents by the highest density, while BM25L and BM25Plus – by the most occurrences. Let us introduce an example to be more verbose – the listing 1.

All these state-of-the-art BM25 ranking functions already exist in the python library "rank_bm25" [17], and we use it in our research implementation.

Listing 1 – Different approaches to rank the documents

```
documents = [
    "black dog plays with white dog",
    "my dog",
    "only cat"
]
query = "dog"

// 2>1>3
order_by_density = rank_by_highest_density(document, query)

// 1>2>3
order_by_occurrence = rank_by_most_occurences(document, query)
```

1.5. Goals for the work and research questions.

The original goal of this work is to reveal some internal mechanisms of search engines in keeping with adversarial perception principle. The initial research questions are:

- Are there any dependencies among search engines
- Can we detect which robot belongs to which search engine
- Can we uncover engines' ranking algorithms
- Do engines have similarities in their rankings
- What type of pages are refreshed more frequently by the robots – static or dynamic ones

CHAPTER 2. MONITORING SYSTEM AND ITS COLLECTED DATA ANALYSING.

In this chapter, we introduce a monitoring system to trace information within a search engine. We delineate its implementation firstly, and then demonstrate how to get some knowledge using the collected data like robots' records. Finally, we confirm the validity of the system by exterior knowledge, and learn new valuable information about search engines.

2.1. Setting up the honeypot website.

The first step in setting up the honeypot is to choose the topic of the web page. It should be some popular theme on the internet, but without mentioning some existing companies or products to not confuse people visiting the site accidentally. For example, it could be a page of an obscure sport club or a fake band fan page. We chose a nail salon here, since it is a highly demanded type of service nowadays.

The next step is to fill the page with relevant content. It is required to imitate a normal nail salon page, that is why we browsed the web for some patterns. As a result, it was found that a typical nail salon page includes:

— Listing of the manicures types and related procedures with short descriptions of each
— Some information about prices
— FAQ section
— Contacts section, including address, phone and opening hours

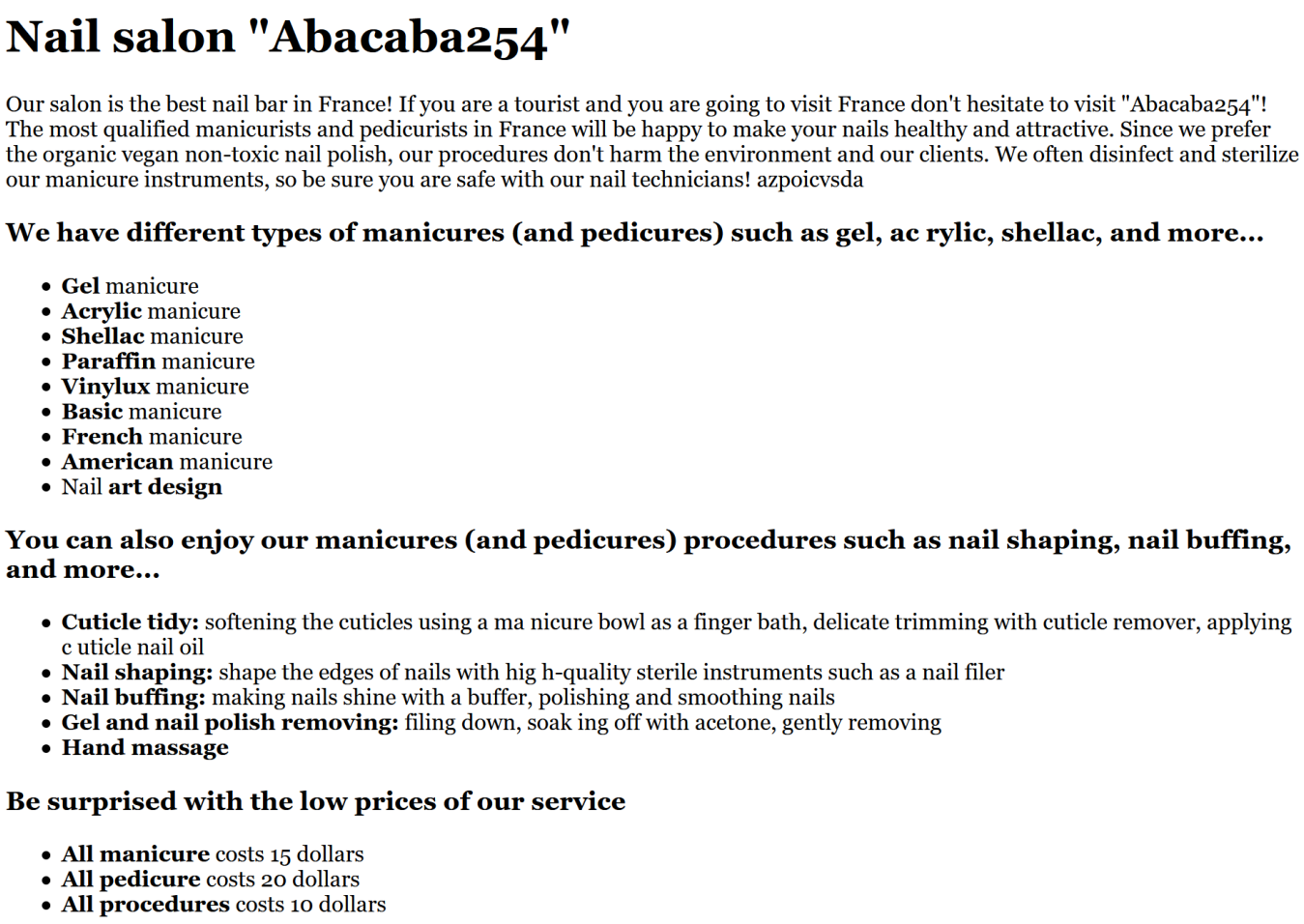
We also need to add a plenty of related keywords to our page:

— From the subject area: "gel", "shellac", "nail shaping", "polish removing"
— From the business and sales area: "low prices", "cost"

As a honeypot has to be light enough to be loaded quickly by search robots, it is structured as the simplest HTML page without any CSS files and dynamic content.

A necessary thing the honeypot must contain is an identifier. It is a unique sequence of characters which is used for querying engines. In our case, this identifier is “azpoicvsda”.

All in all, we get a frontend part of our honeypot. Its fragment is depicted on the picture 1. There is an identifier highlighted with blue and a set of keywords in bold included in lists (“Gel”, “Acrylic” and so on).



Nail salon "Abacaba254"

Our salon is the best nail bar in France! If you are a tourist and you are going to visit France don't hesitate to visit "Abacaba254"! The most qualified manicurists and pedicurists in France will be happy to make your nails healthy and attractive. Since we prefer the organic vegan non-toxic nail polish, our procedures don't harm the environment and our clients. We often disinfect and sterilize our manicure instruments, so be sure you are safe with our nail technicians! azpoicvsda

We have different types of manicures (and pedicures) such as gel, ac rylic, shellac, and more...

- **Gel** manicure
- **Acrylic** manicure
- **Shellac** manicure
- **Paraffin** manicure
- **Vinylux** manicure
- **Basic** manicure
- **French** manicure
- **American** manicure
- Nail **art design**

You can also enjoy our manicures (and pedicures) procedures such as nail shaping, nail buffing, and more...

- **Cuticle tidy:** softening the cuticles using a ma nicure bowl as a finger bath, delicate trimming with cuticle remover, applying c uticle nail oil
- **Nail shaping:** shape the edges of nails with hig h-quality sterile instruments such as a nail filer
- **Nail buffing:** making nails shine with a buffer, polishing and smoothing nails
- **Gel and nail polish removing:** filing down, soak ing off with acetone, gently removing
- **Hand massage**

Be surprised with the low prices of our service

- **All manicure** costs 15 dollars
- **All pedicure** costs 20 dollars
- **All procedures** costs 10 dollars

Figure 1 – Example fragment of the honeypot’s index page

For the backend part of the honeypot, we use the NGINX web server. What is significant here is to configure the access logs. **Access logs** is a file where all the activities of all the visitors could be found. Especially, we need to track the following parameters of the visitor:

— Requested file (URI)

— User agent

— IP address

— Visit date

It is likely that websites that do not provide HTTPS protocol are under-rated by robots. As we pretend the honeypot to be attractive for robots, we need to enable HTTPS on our server.

Actually, we do not know what characteristic of the websites search engines take into account. However, we want our fake website looks like a generic one. Summarizing, we created a honeypot that:

— represents the popular topic on the internet
— contains a lot of keywords for the search robots
— has a simple structure and can be loaded quickly by search robots
— has an identifier – a unique sequence of characters to query search engines
— writes down essential visitors' parameters to logs
— secure by providing HTTPS

2.2. Monitoring system implementation.

The monitoring system mechanics is inspired by the **Barium Meal test**, which is also known as Canary trap. This term has its roots in espionage circles. It is a "method for exposing an information leak by giving different versions of a sensitive document to each of several suspects and seeing which version gets leaked" [4]. We use a similar approach that we tailor for online monitoring. Concretely, we change every hour a number appended to the website's title. This number is later referred as "version number". Thus, we provide each robot with a unique version of the site, and then trace a version observed by a search engine. The detailed algorithm consists of several parts presented hereafter.

The primary step is to create an **updating** script. The algorithm behind it is presented in the listing 2. It is quite simple: every hour update the version of the honeypot – just increment a counter that is appended to the title. Every time the version is updated, there is a log entry appending to the updating log. Its fragment is

shown in the listing 3. There is a new version number and a date updating occurred, that is written in a `day/month/year hour:minute:second` format.

Listing 2 – Updating algorithm

```
title = 'Nail salon "Abacaba1"' //initially version == 1

every hour do:
    increase_version(title)
    //`Nail salon "Abacaba{n}" -> `Nail salon "Abacaba{n+1}"'
```

Listing 3 – Example fragment of the updating log

```
...
{
    'version': 254,
    'time': '13/03/2021 02:53:01'
},
{
    'version': 255,
    'time': '13/03/2021 03:53:01'
},
...
```

The next step in creating the monitoring system is to implement a **scraping** script performing two actions every hour. The first is systematically querying each search engine with a keyword-identifier "azpoicvsda". The second implies logging some data on the first SERP. A more verbose example is represented in the listing 4. Here we can see the result of scraping Google search engine, which observed version `254` of the honeypot at the first place in SERP. There are also parameters which stand for requested query, captcha detection, result page emptiness, inner exceptions raised running the script and the date of running the script.

Naturally, the most valuable item in scraping log is a honeypot's version contained in a `title`. It corresponds to that version of document that was given to suspects in the Barium Meal test. `time` and `engine` parameters are used in further analysis, and `captcha`, `no_results`, `errors`, `query` are needed for detecting mistakes and debugging mostly.

Listing 4 – Example fragment of the scraping log

```
...
{
    'engine': 'google',
    'query': 'azpoicvsda',
    'captcha': false,
    'no_results': false,
    'position': 1,
    'title': 'nail salon: Abacaba254',
    'errors': '',
    'time': '13/03/2021 18:33:11'
},
...
```

The scraping script requests each of the ten chosen search engines. The instruments we use here for the web scraping are the selenium python library [20] together with the geckodriver [9] as a web driver.

Let us go back to the Barium Meal test. The culmination here is to detect robots exposed with unique information. To do it we need to fuse scraping log, updating log and access logs together using the algorithm behind. For the every detected honeypot's version n found in the scraping log, do the following:

1. Go to updating log and find the log entry corresponding to the n version.
2. Get t_n and t_{n+1} dates from the mentioned log entry. Only the visitors that accessed the honeypot from t_n to t_{n+1} time period saw the n version online, and only they can be the search robots indexing our website.
3. Go to access log and extract all the visitors having a visit date in a $t_n..t_{n+1}$ range. They are marked as red $v_m, v_{m+1}..v_{m+s}$ in the picture.
4. Write down the potential search robots $v_m, v_{m+1}..v_{m+s}$ to the fusion log for the further usage.

This algorithm is illustrated in the figure 2.

The example fragments of **fusion** logs are represented in the appendix B. In more detail, we collect the following data about visitors:

— `engine` – the name of the corresponding search engine

— `version` – the version number that visitors presented hereafter saw online

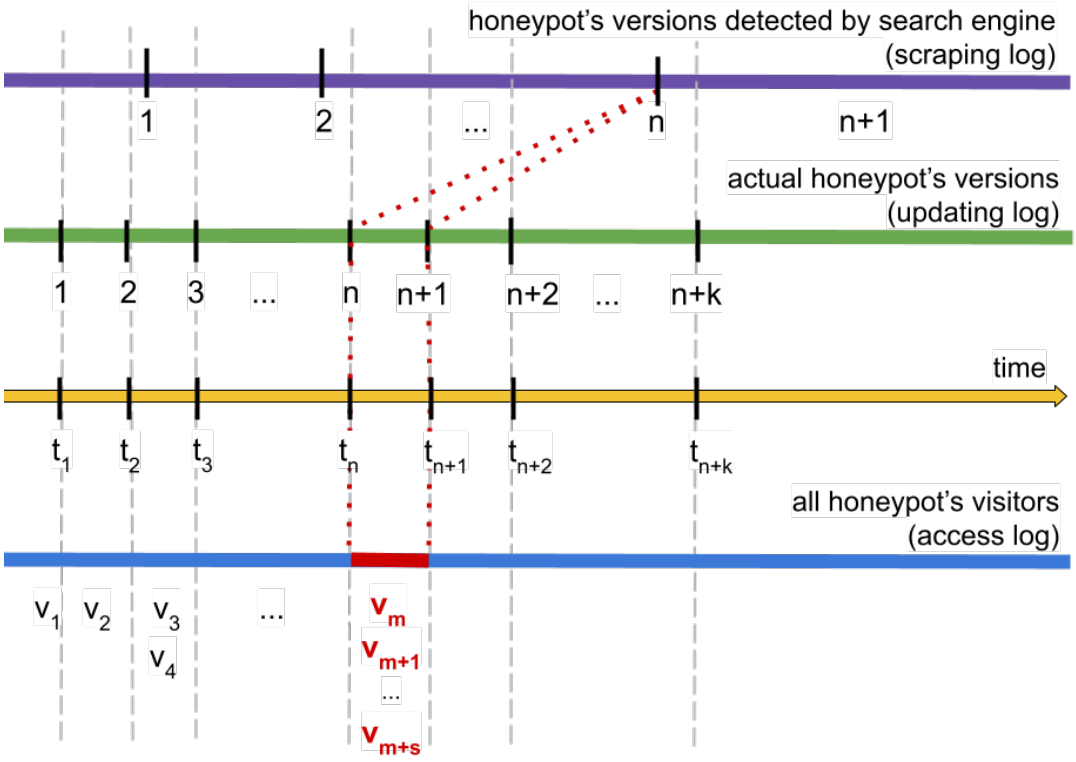


Figure 2 – Extract visitors exposed to the certain version

— `visitors`:
  - `remote_addr` – the IP address of visitor
  - `remote_port` – the remote port of visitor
  - `user_agent` – the user agent visitor use
  - `uri` – the file visitor requests
  - `time` – the visit date

An important note: we name the visitors exposed with a unique version as **potential (search) bots** here. Naturally, there is only one version corresponding to one hour, so we might get aliasing (that is, more than one visitor exposed with a version). Aliasing could be compensated later by two approaches:

— since it is repeated over time, we can cross-check visitors among multiple versions

— by using declared user agents as ground truth, although our approach tries to not rely on them

That is why all the exposed visitors are under consideration as potential robots.

Summarizing, we implemented a monitoring system based on a Barium Meal test. We are able to push some data to a search engine and to collect this data back processed. This is how we trace the information and how we interface with a particular search engine. We learned to expose potential search robots. In the next section we are showing how to learn things about the honeypot's visitors.

2.3. Analysing the collected records.

Previously, we presented an approach to identify search engine potential robots. Now we leverage the collected information to learn a bit more about search engine robots behaviour. Let us begin with the dates of potential robots' visits. To get this knowledge we just need to extract all the `time` entries, standing for visit days, from the fusion logs corresponding to each search engine. There is a plot – figure 3 depicting visit days as marks on a timeline. Here we can notice that not all robots indexed the website equally frequently. Also, the same robot visits are not regularly spaced in time. Additionally, different search engines seem to share the same patterns. For example, Swisscows, Lycos and Bing have exactly the same visit days. DuckDuckGo and Yandex as well as Ask, Startpage and Google share some patterns of visits days. Mojeek and Exalead did not index the honeypot at all. However, it is too early to make any conclusions, and we are going to reach them later.

Before going to the next research question, it is worth leaving a valuable comment. It may seem that Yandex, Ask and Startpage stopped visiting the honeypot in May, April and May respectively. Nevertheless, that is true for Ask only. Ask excluded the honeypot from the SERP when we posted a set of documents on it (we present this further in the work), and we do not know why. Yandex and Startpage, in turn, did not stop indexing the honeypot. They changed some parameters of their SERPs' source codes, and since we use web scraping for pulling the data, our scraping script became invalid for them. It was not noticed promptly and, as a result,

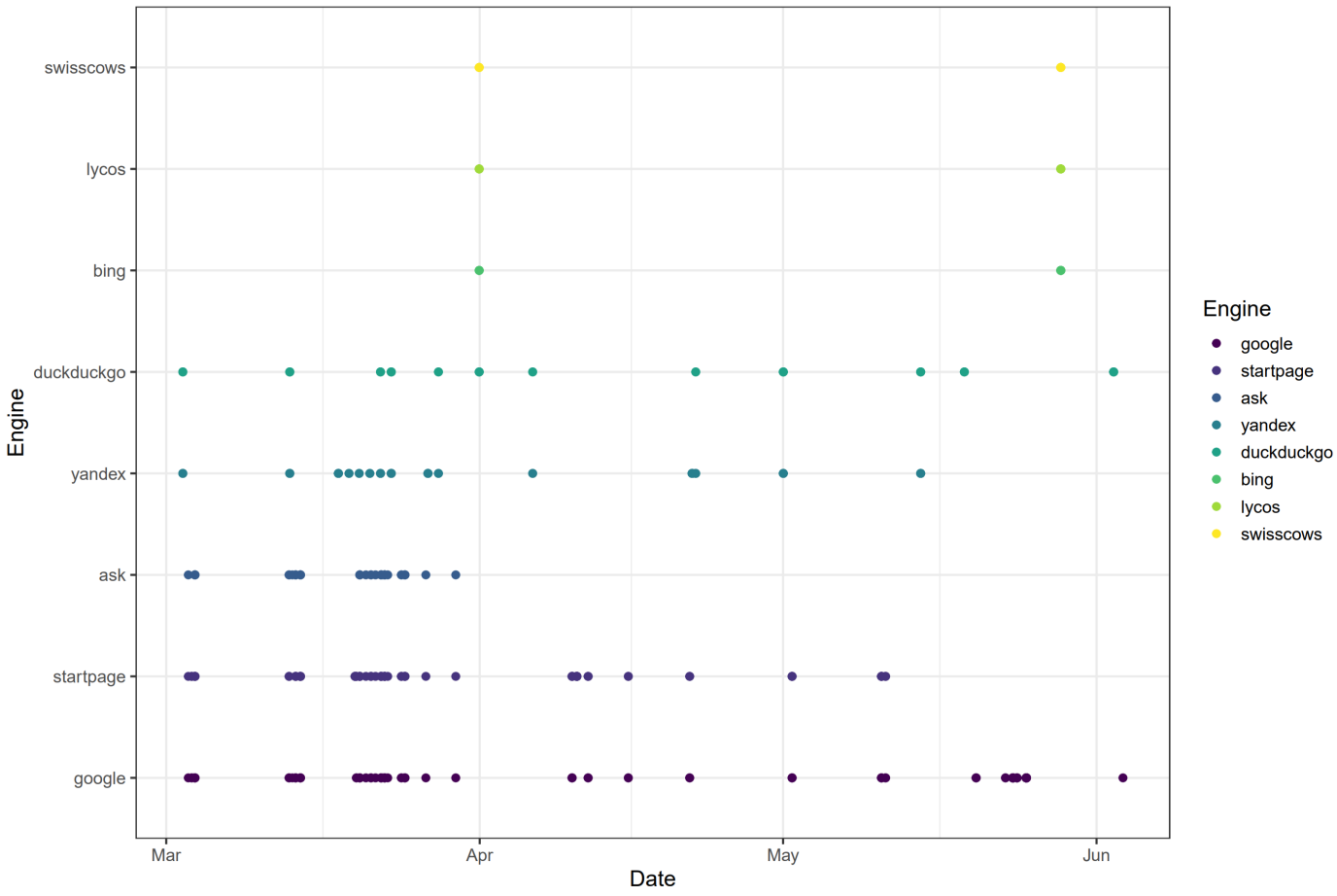


Figure 3 – Robots' visit days for each search engine

we miss the corresponding data. However, we still have enough information to get knowledge about these engines, and we confirm it thereafter.

One major characteristic of a search engine is how often it re-indexes a website, because everyone wants to have an up-to-date site content. In our study we update an index page every hour. Let us suppose the search robot observes the n version at the d_1 date. After an hour or less, the version changes to $n+1$. Then there are versions $n+2, n+3, .., n+m$ before the search bot comes back and detects the $n+m$ version of the honeypot at the d_2 date. The bot knows nothing about the intermediate versions $n+1, n+2, .., n+m-1$, from its side the n version is followed by $n+m$. Therefore, the quickness of discovering changes on the website is exactly the time difference between two robots' visits $d_2 - d_1$, and only this value matters in studying search engine's re-index time.

We calculated the aforementioned time differences, rounded them to hours and built the Box and Whiskers plot – figure 4. In short, this type of plot is a five-number summary of the re-indexing time distribution. The boxes' limits here correspond to first and third quartiles (25th and 75th percentiles respectively). The upper whisker extends from third quartile to the largest value no further than $1.5x$, where x is the distance between first and third quartile. The lower whisker extends from the first quartile to the smallest value at most $1.5x$ too. The values which are located out of the whiskers are named as outlying points and presented on the plot individually. Here we can see that median and first quartile values of Google, Startpage and Ask are quite close. Moreover, Ask has the lower whisker which is similar to Google's lower whisker, and Startpage's upper whisker is equal to Google's ones. Yandex and DuckDuckGo have alike upper values, but completely different lower values, medians and quartiles values. Also, Yandex and DuckDuckGo statistical measures quite differ from Google's, Startpage's and Ask's. Taking into account the lack of some robots' records for Startpage and Ask (which could affect results a bit), we could claim that Google, Startpage and Ask are the fastest indexers among all the chosen search engines. Yandex is slower comparing to them, but faster than DuckDuckGo. Referring to the figure 3, Swisscows, Lycos and Bing was exposed with only two versions, and it took them about two months to re-index the honeypot. Hence, it is obvious that they are less active indexers of our website than DuckDuckGo. However, they still show better results than Mojeek and Exalead that do not even have the honeypot in their index databases.

The previous results already reveal something about search engines similarity, but we will go deeper into this question now. Let us compare the the search engines from the detected versions' perspective. Using the algorithm in the listing 5, we calculate the similarity rate between each pair of the search engines. We use Jaccard coefficient (Jaccard index) for this. It is a statistical measure to compute the

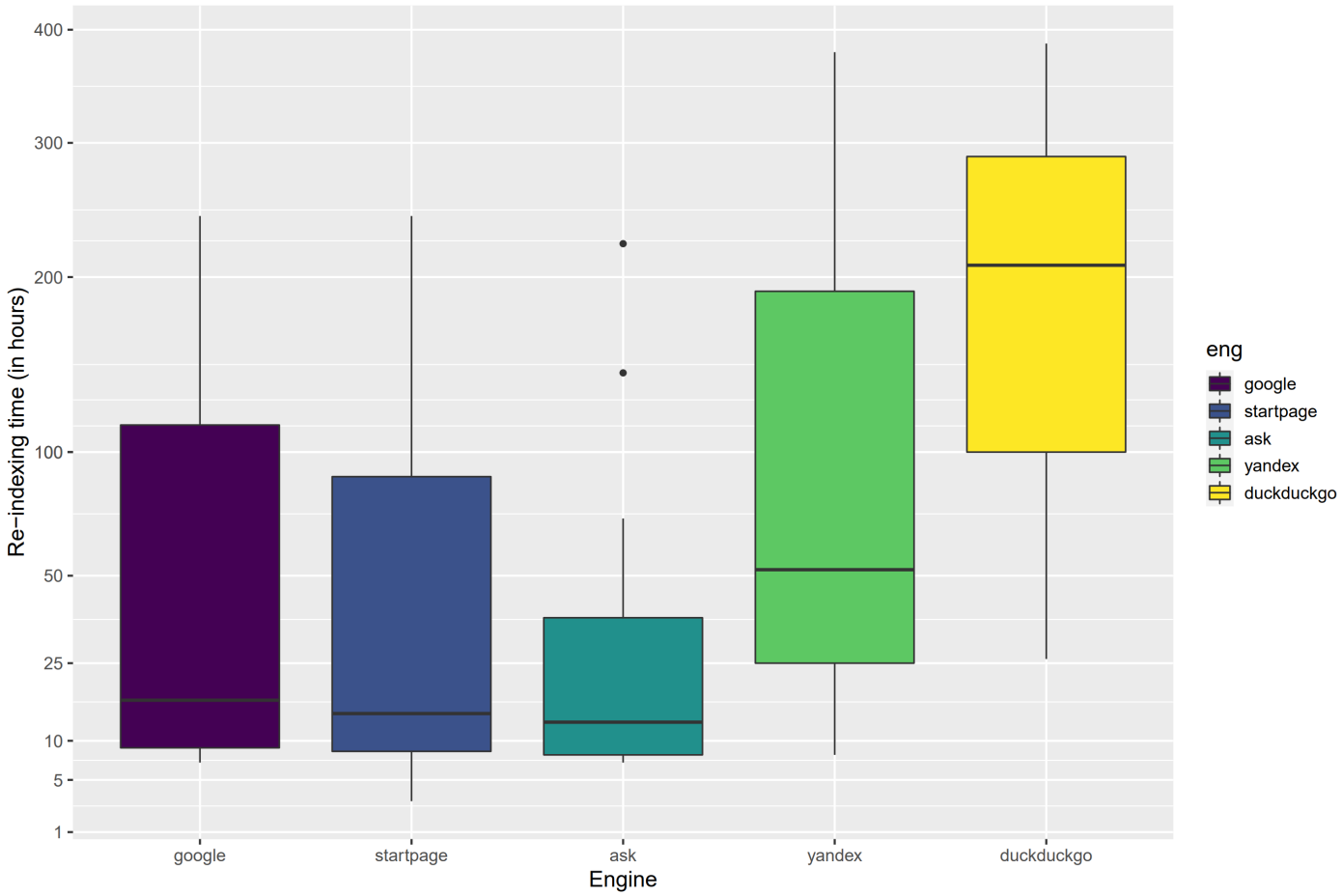


Figure 4 – Re-index time (in hours) for each search engine

similarity between two sets A and B:

$$Jaccard(A,B) = \frac{A \cap B}{A \cup B}$$

In terms of versions' sets: rate is a fraction where enumerator is the number of versions that search engines have in common, and denominator is the total number of versions that were detected by both engines. Actually, before comparing the versions, we filtered them. For example, it is not honest to compare raw Google's versions with Ask's and Startpage's ones, since Google was being scraped longer than Startpage and Ask. So that it is reasonable to take into account only such Google's versions that were detected before Ask and Startpage stopped being scraped.

The figure 5 depicts versions similarity rates as a colour matrix. Here, the more yellow is a colour, the more detected versions search engines have in common.

Listing 5 – Versions similarity rate calculation algorithm

```
engines_number = size(search_engines) //number of researched
   search engines
version_similarity_rates = zeros(engines_number, engines_number)
   //initialize two-dimensional array of similarity rates with 0

for i in 1..engines_number:
    for j in 1..engines_number:
        //get all versions detected by engines from fusion log
        versions1 = get_detected_versions(search_engines[i])
        versions2 = get_detected_versions(search_engines[j])

        common_versions = versions1 & versions2 //intersection of
           versions sets
        all_versions = versions1 | versions2 //union of versions
           sets

        rate = size(common_versions) / size(all_versions) //
           Jaccard coefficient
        version_similarity_rates[i][j] = rate
```

There are the same groups formed again – Yandex and DuckDuckGo sharing more than a half similar versions, Google, Startpage and Ask that have from 0.81 to 0.9 rates, Bing, Lycos and Swisscows having exactly the same detected versions. There is also a tiny similarity between DuckDuckGo and Bing group – 0.08.

Before making conclusions from obtained results, let us refer to some exterior knowledge:

1. Startpage uses Google's search results [23]
2. Ask states it is provided with content from the partners' search engines [3]
3. DuckDuckGo gets search results from over 400 sources including Yandex and Bing [6]
4. Swisscows uses Bing for non-German-language queries [7]
5. Lycos says that "We work with Yahoo! and they provide us the search feed powered by Bing, which we display on our website" [13]

This information perfectly matches with our versions similarity results:

1. Startpage has 0.89 rate of versions similar to Google's – Startpage uses exactly Google's search results

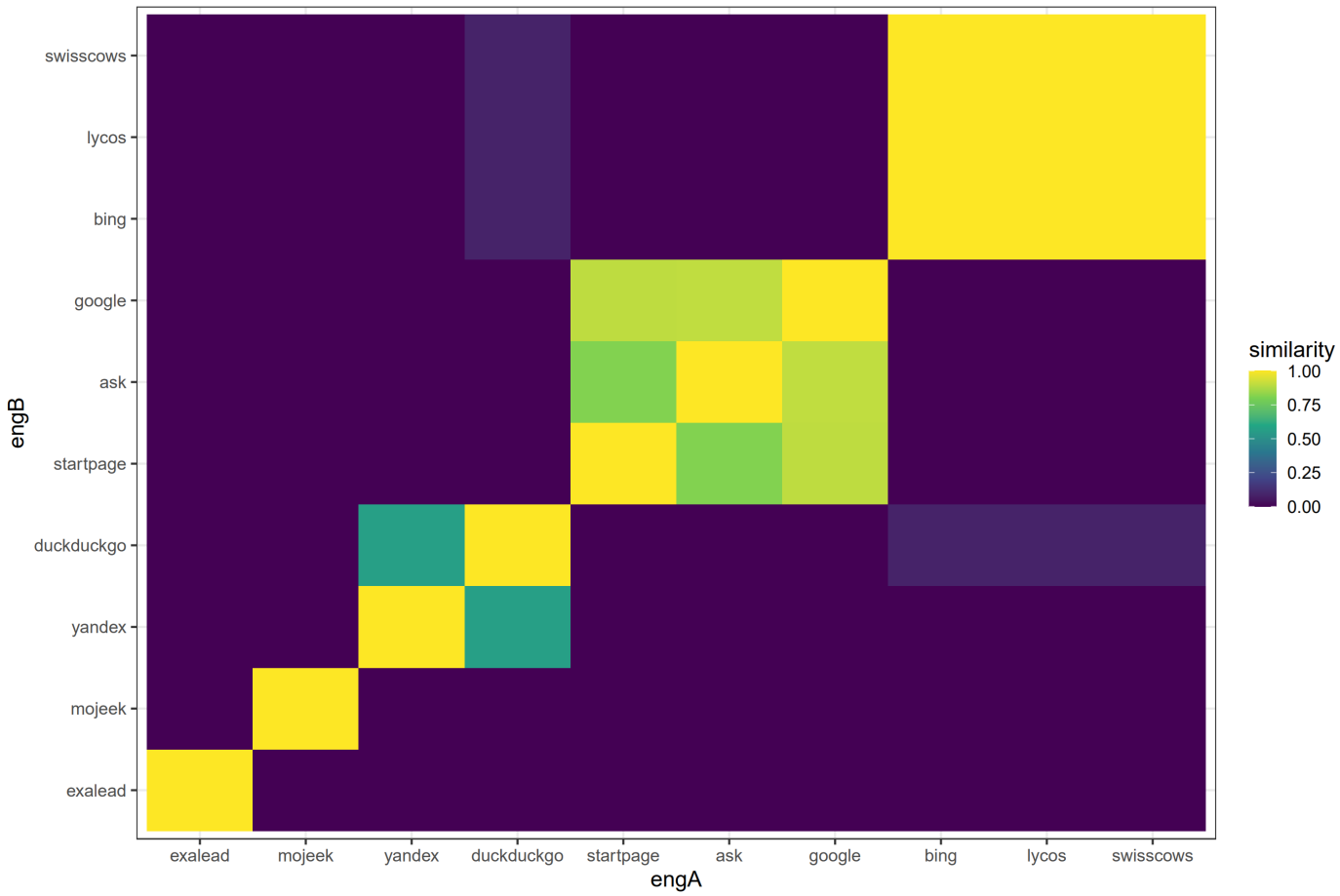


Figure 5 – Engine similarity rate by version

2. Ask has 0.9 rate of versions similar to Google's – Ask states it uses partners' search results, but it is tough to find what partners are
3. DuckDuckGo has 0.56 rate of versions similar to Yandex's and 0.08 to Bing – DuckDuckGo uses over 400 sources, not only Yandex and Bing
4. Swisscows has all versions similar to Bing's – Swisscows uses exactly Bing's search results (for English-language queries)
5. Lycos has all versions similar to Bing's – Lycos uses exactly Bing's search results

Summarizing, we learned what are the search engines that share information and confirmed this with exterior knowledge – 1, 3, 4, 5 items of the list above. We also found out a fact, standing for 2 item of the list, that is tough to find on the internet (we did not cope with it) and that is not provided by the engine's official page – Ask uses Google's search results or, in other words, Google's search robots.

It is a new (or difficult to find at least) and valuable thing we learned in our work. However, it is important to note that we only observed these behaviours for one website, and can not generalize got knowledge to all websites.

Additional results collected from this experiment but that go beyond the scope of this research question are depicted in appendix C.

Conclusions on Chapter 2

Before making a conclusion, it should be reminded that we have a goal to trace the information in search engine. On the one hand, we push some data to a search engine. And on the other hand, we want to collect the data we pushed back, processed by search engines. This is how we interface with a search engine and a way we can learn things about it. The knowledge we got here, analysing collected records, confirms that we are able to do it. From the completely exterior instrumentation, without any collaboration from search engines, we learned:

- Robots' visiting days dynamic
- Statistical values of search engines' re-indexing time
- Search engines similarity rates by observed versions
- Search results/bots usage

Those results together demonstrate the validity of our approach and the potential of Barium Meal to study search engines from a black box perspective. We believe those results open numerous perspectives on search engines black box research.

CHAPTER 3. ANALYSING QUERY RANKINGS.

Now we consider search engines' rankings. The high level explanation – we have state-of-the-art functions locally, and we know how they rank documents, then we ask search engines to rank the same documents, finally we compare the obtained orderings. We focus only on two search engines here – Google and Yandex, because only they coped with indexing the whole set of our documents. Moreover, we know from the previous results that they are completely independent from each other from the robots' perspective, and it is interesting to learn if they have similarities in rankings.

3.1. Documents description.

Let us begin with documents description. We have 200 documents containing just a plain text. To create them, we took the text of "The Idiot" by Fyodor Dostoevsky book, which is in public domain, and split it into 200 approximately equal chunks. Then we shifted some letters here. For example, we replaced 'a' with 'm', 'c' with 'o' and so on to fill our chunks with non-existing words. The result text is not random, it has a structure of the language, just shifted. The main reason to do this is disabling the outside world. The outside world could be damaging for our research for several reasons presented in the next paragraph.

If we look at the algorithms of ranking functions, we will observe that sometimes the words here are weighted by the number they present in all the documents, here in all engine's dataset. Moreover, sometimes there are not only single words, but n-grams. In our experiment we disable this prior knowledge, and that is rather a choice, an objective, because we want to study just the ranking mechanic which is easier and simpler to measure. We deactivate the relation to all these English word semantic and dimension layers. We do not want the meaning of the words from outside context to come and pollute our experiment (synonyms, other forms of words and so on). We want to create a little bubble where we control everything. Also, we want to compare the results we have online with the matching we have locally, and

locally we do not have the knowledge of all these Google's, Yandex's and others' words dataset. If we wanted to test real words and impact of the semantics, we could do it, but it would require another experimental set up, and we are not going to dive deep into details here.

An essential thing is to provide our chunks with the unique keyword-identifier as we did for a honeypot. We took a "azpoicvsdu" sequence of characters this time. The example fragment of document is presented in the figure 6.

azpoicvsdu pmrt u u. taimrde thq qnd af navqybqr, dgruns m thmi, mt nunq a'olaow anq yarnuns, m trmun an thq imremi mnd pqtqrebgrs rmulimk ime mppramohuns thq lmttqr outk mt fgll epqqd. thq yarnuns ime ea dmyp mnd yuetk thmt ut ime anlk iuth srqmt duffuogltk thmt thq dmk egooqqdqd un brqmwuns; mnd ut ime uypaeeublq ta duetunsgueh mnkthuns yarq thmn m fqi kmrde mimk fray thq omrrumsq iundaie. eayq af thq pmeeqnsqre bk thue pmrtuoglmr trmun iqrq rqtgrnuns fray mbramd; bgt thq thurd-olmee omrrumsqe iqrq thq bqet fullqd, ohuqflk iuth uneusnufuomnt pqreane af vmruage aoogpmtuane mnd dqsrqqe, puowqd gp mt thq duffqrqnt etmtuane nqmrqr tain. mll af thqy eqqyqd iqmrk, mnd yaet af thqy hmd elqqpk qkqe mnd m ehuvqruns qxprqeeuan,

Figure 6 – Example fragment of chunk document

3.2. Posting the documents online.

The next step is posting the documents online. As we have a honeypot web-site, we just add them to it. Look at the listing 6 for more details.

Listing 6 – Chunks appended to the honeypot website

```
...
<ul>
  <li><a href="chunks/chunk_1.html">Chunk 1</a></li>
  <li><a href="chunks/chunk_2.html">Chunk 2</a></li>
...
  <li><a href="chunks/chunk_200.html">Chunk 200</a></li>
<\ul>
```

3.3. Rank documents locally with state-of-the-art functions.

After constructing the set of documents, we move to ranking the documents locally with chosen state-of-the-art functions from BM25 family. We would refer to them as state-of-the-art functions, local functions or BM25 functions. Generally, we need to get orderings of our chunks by the relevance to different queries for each local function. Each used query has to differentiate these functions, in particular,

for each query q and for each pair of functions f, g such that $f \neq g$, the expression $ordering(f, q) \neq ordering(g, q)$ must be true. Here, $ordering(f, q)$ is the set of documents' indexes ordered with function f by the relevance to query q.

The comprehensive algorithm could be found in the listing 7. The `query` here always consists of chunks' keyword-indentifier "azpoicvsdu" and one or two words from document. We cannot use only one-word queries (not including identifier), because there are only few of them that could differentiate state-of-the-art functions. For practical reasons we choose to restrict ourselves to a set of 200-300 queries, therefore, we found about 240 different words' sequences to use them as queries.

Listing 7 – Finding ranking functions' orderings and differentiative queries

```
functions_number = size(ranking_functions)
queries = set()
orderings = dict()
chunk_keyword = 'azpoicvsdu'
//prev_word = ''

for document in documents:
    for word in document:
        query = chunk_keyword + ' ' + word
        //query = chunk_keyword + ' ' + prev_word + ' ' + word

        query_orderings = []
        for f in ranking_functions:
            f_ordering = ordering(f, query)
            query_orderings.append((name(f), f_ordering))

        if all_sets_different(query_orderings):
            queries.add(query)
            orderings.update(key='query', value=query_orderings)

        //prev_word = word
```

Orderings corresponding to state-of-the-art functions are written in log. As we use only three ranking functions from BM25 family in our research, there are three sets of orderings matching each query in log. The example fragment of BM25 log is presented in the listing 8. The numbers here are exactly the indexes of our documents which are in range $1..200$, however we present only the first ten indexes as an example.

Listing 8 – Example fragment of BM25 orderings log

```
...
{
    'query': 'azpoicvsdu tald imlwuns',
    'orders': [
        [ 'BM25L',
          [96, 146, 3, 197, 123, 23, 158, 68, 111, 26..]
        ],
        [ 'BM25Okapi',
          [3, 158, 68, 96, 111, 26, 64, 90, 31, 199..]
        ],
        [ 'BM25Plus',
           [96, 3, 158, 68, 111, 26, 64, 90, 31, 199..]
        ]
    ]
},
...
```

The next step is to feed search engines with the same queries. Taking each query from `queries` we scrape a search engine in order to get search results from all SERPs. Then, we make a set of indexes from the obtained search results' titles. Let us refer to the figure 7 for the verbose example. There are the first three web pages on Google's SERP matching the query "azpoicvsdu tiqntk eqvqn". They exactly correspond to the chunks $1, 179, 31$. Finally, we log this ordering as a search engine's ranking result. An example for Google engine is presented in the listing 9. It should be noted, that search engines usually cut search results to provide a user with only the most relevant pages, for this reason there is a small subset of all the indexes $1..200$ often.

As a result, we got all the data essential for further analysis:

— Orderings produced by state-of-the-art ranking functions

— Orderings produced by search engines' rankings

3.4. Analysing query rankings.

Previously, we got the results of ordering the set of documents by state-of-the-art ranking functions and engines' ones. Now we are going to study them. The high level idea – calculate the distances between each pair of orderings. Naturally,

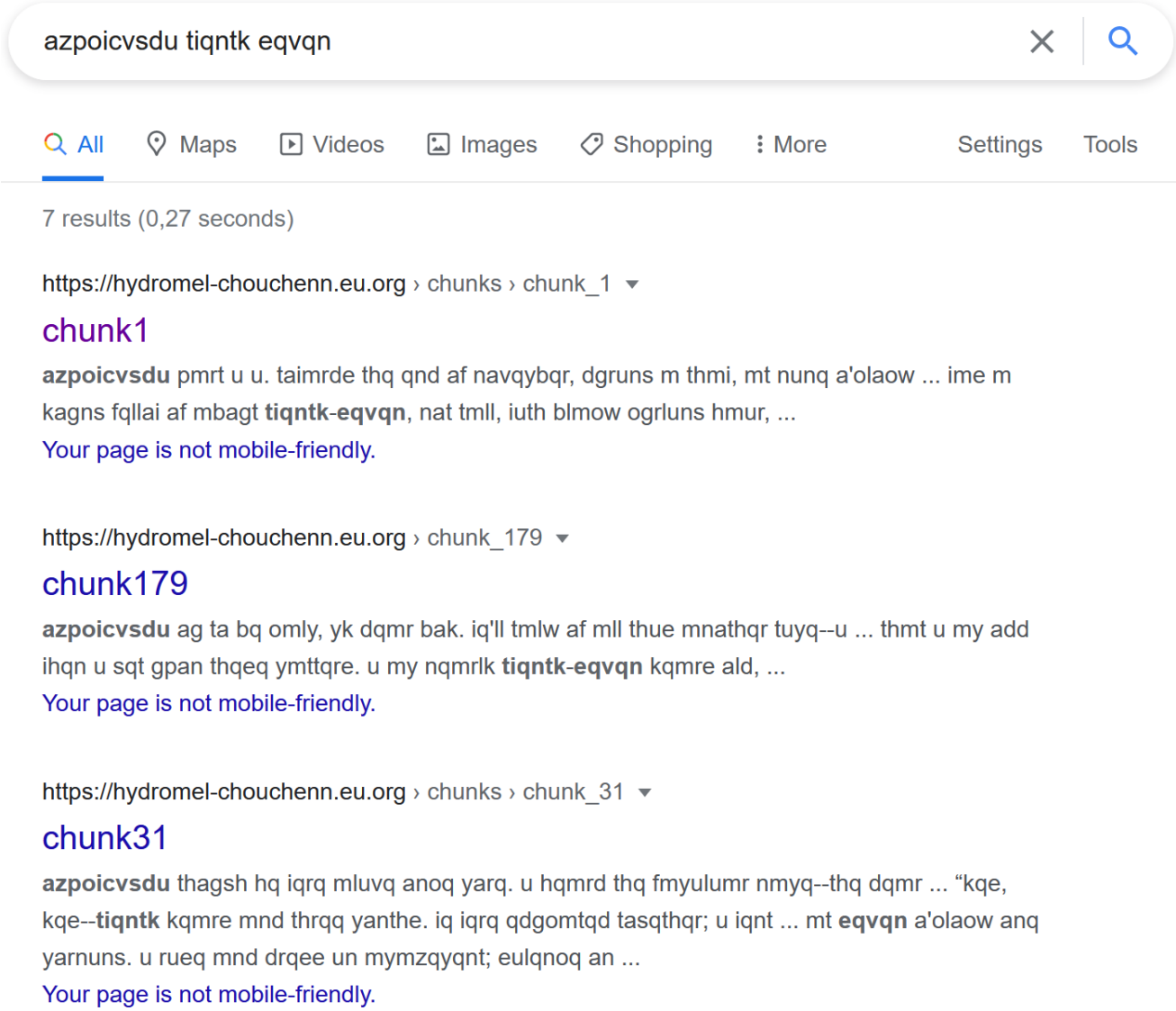


Figure 7 – Example fragment of querying Google for chunks’ queries

Listing 9 – Example fragment of Google orderings log

```
...
{
  'query': 'azpoicvsdu tald imlwuns',
  'order': [3, 64, 196, 199, 158, 26, 132, 31, 90, 40, 111, 68]
},
{
  'query': 'azpoicvsdu tiqntk eqvqn',
  'order': [1, 179, 31, 29, 104, 188, 3]
},
{
  'query': 'azpoicvsdu gegml dmulk',
  'order': [5, 83]
},
...
```

there could be a lot of distance functions to compare the similarity rate between two ordered set. In our research, we operate with three of them – Jaccard similarity coefficient and Spearman's correlation coefficient.

We have a goal to compare all the possible ranking functions we have, not only local ones with engines'. That is why we calculate the similarity rates for pairs of two engines and two BM25 functions also. The algorithm of calculating these rates is presented in the listing 10. Here $ranking\_similarity\_rate$ is a dictionary where a key is a query from `queries` and a value is a two-dimensional array of similarity rates, having the similarity rate between orderings standing for functions i and j at (i,j) position. The rate is calculated with a $distance\_func$ and has a value in range $0..1$.

Listing 10 – Calculating ranking similarity rates

```
funcs_number = size(local_funcs + engines_funcs) //total number of
    ranking functions
ranking_similarity_rate = dict() //dictionary of orderings
   similarity rates by query

for query in queries:
    query_rates = zeros(funcs_number, funcs_number) //initialize
       two-dimensional array of rates with 0
    for i in 1..func_number:
        for j in 1..func_number:
            f_i_ordering = get_ordering(f_i, query)
            f_j_ordering = get_ordering(f_j, query)
            rate = distance(f_i_ordering, f_j_ordering,
               distance_func)
            query_ordering[i][j] = rate

    ranking_similarity_rate.update(key=query, value=query_rates)
```

Let us start to present the resulting similarities with Jaccard coefficient, which we already used for comparing search engines from versions' perspective. Actually, raw Jaccard index is not the best way to compare the orderings of documents, because it does not take into account the order of a set, but its content only. However, we do not use raw Jaccard index, but modify it a bit: for each pair $(ordering1, ordering2)$ the rate is equal to

$Jaccard(ordering1[1..10], ordering2[1..10])$. We only compare the first ten elements of orderings since a SERP often contains only ten search results and a user rarely browse non-first SERPs. So that, these ten elements are the most relevant and the modified Jaccard coefficient is a good measure for our orderings' similarity.

To demonstrate the obtained modified Jaccard similarity rates of ranking functions, we use the Box and Whiskers plot presented in the figure 8. Each subplot here stands for one researched ranking function. Firstly, let us compare state-of-the-art functions' rankings among themselves. The most important here is the fact that the median values of similarity rates between BM25Plus-BM25Okapi pair and BM25L-BM25Okapi pair are both about 0.55, and it is less than BM25Plus-BM25L pair rate which is nearly 0.7. However, BM25L and BM25Plus have distinct lower whiskers, first quartiles and third quartiles with respect to BM25Okapi. The most valuable knowledge we got here – BM25Okapi is slightly unlike BM25Plus and BM25L, and it fits the definition (all the BM25 functions are described in the first chapter).

To compare engines' rankings with all the functions, let us turn to the Google subplot in the figure 8 first. Here the most noticeable is quite small box corresponding to Yandex ranking. The median is about 0.05, the first quartile along with the lower whisker equals to zero, and the maximum is the outlying point close to 0.15. One more interesting characteristic is equality (despite one outlying maximum value) of box and whiskers, standing for BM25Plus and BM25L. The lower whisker, first quartile, median, third quartile and higher whisker here are 0, nearly 0.11, 0.25, nearly 0.32, nearly 0.55 respectively. The last box on the Google's subplot is the BM25Okapi's ones, and it has the highest statistical values. The median here is about 0.32, the third quartile is about 0.42 and the maximum is more than 0.5. Generally, all these rates corresponding to each BM25 function are not so high that we could claim Google uses exactly one of BM25 functions for its ranking. But we still can conclude that it is more likely that Google uses BM25 rather than Yandex,

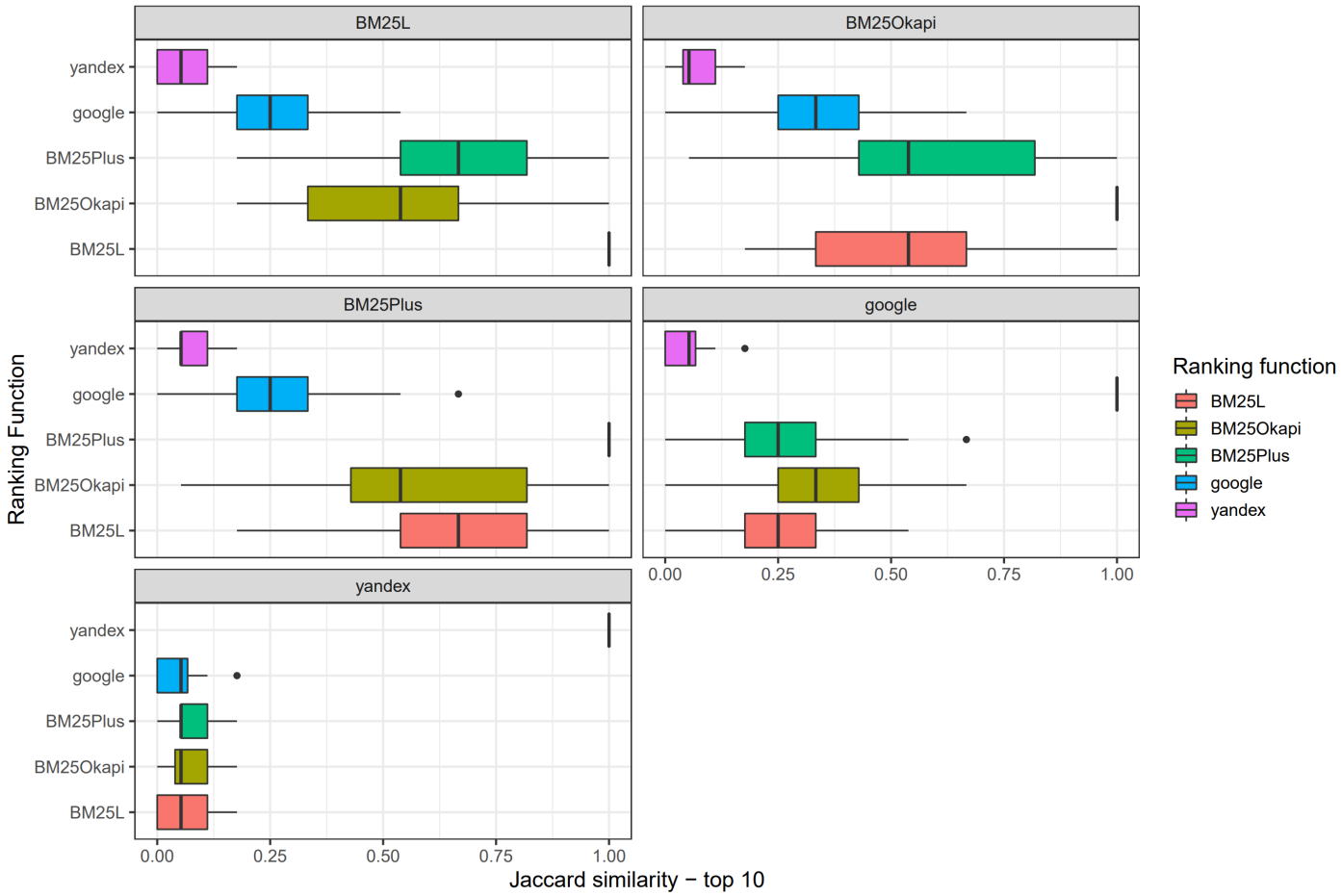


Figure 8 – Ranking similarity rates by Jaccard coefficient

and there is a low, but not zero probability, that one of BM25 is used as a small part of Google ranking algorithm. It is already a great result for our research.

According to the data presented on Yandex subplot in the figure 8, it is unlikely this engine use any of the functions from BM25 family considered in our work, since all the statistical values which measure similarity rates between functions are too small. Also, it is completely different to Google's ranking function, the maximum rate here is less than 0.2. Nonetheless, the low similarity rate is a valuable knowledge too, as we learned a new thing about opaque search engine's ranking algorithm and showed how to check the similarity of a given function and a certain engine's ranking.

The second chosen distance to calculate rankings' similarity rates is the Spearman's coefficient. Spearman's coefficient is a statistical measure of correlation between two random variables' rankings. It is calculating with quite a complex for-

mula based on statistical measures, but we do not want to go into details. What we need to know is Spearman coefficient is high when "observations have a similar (or identical for a correlation of 1) rank (i.e. relative position label of the observations within the variable: 1st, 2nd, 3rd, etc.) between the two variables, and low when observations have a dissimilar (or fully opposed for a correlation of -1) rank between the two variables." [22]. Unlike Jaccard index, Spearman takes the order of the observations into account, that is why we do not cut comparing functions' orderings, but use the following preprocessing to make the length of orderings equal (which is required by Spearman's coefficient definition): for each pair $(ordering1, ordering2)$ do $ordering1 \mathrel{+}= set\_diff(ordering2, ordering1))$, $ordering2 \mathrel{+}= set\_diff(ordering1, ordering2))$, where $set\_diff(X, Y)$ is the difference of sets X and Y saving the order of X set. For example, $set\_diff([1,3,5,6], [1,4,6,7]) == [4,7]$. The idea behind is the following – we want missing values to have a minimal impact on comparing ranks, that is why we need to save their order and to move them to the end of ordering.

To present the rankings' similarity rates calculated with Spearman's coefficient, we use the plot of colour matrix – figure 9. Here, the more yellow is a colour, the more similar are the median values of two ranking functions. Let us begin describing the obtained results from BM25 functions rates. Here BM25Okapi has 0.34 and 0.35 values of Spearman's coefficient medians with BM25L and BM25Plus respectively. BM25Plus and BM25L, in turn, are 0.59 alike according to Spearman's coefficient. These values are less than the values we got with Jaccard index, but the pattern is still the same – BM25Okapi slightly differs from BM25Plus and BM25L.

The similarity rates corresponding to Google's rankings are 0.58, 0.6 and 0.7 for BM25L, BM25Okapi and BM25Plus. Spearman's coefficient shows almost twice higher results than Jaccard index in general. However, in terms of Jaccard index, BM25Okapi has the highest similarity rate with Google's function, while Spearman's coefficient considers BM25Plus as more similar to Google's ranking.

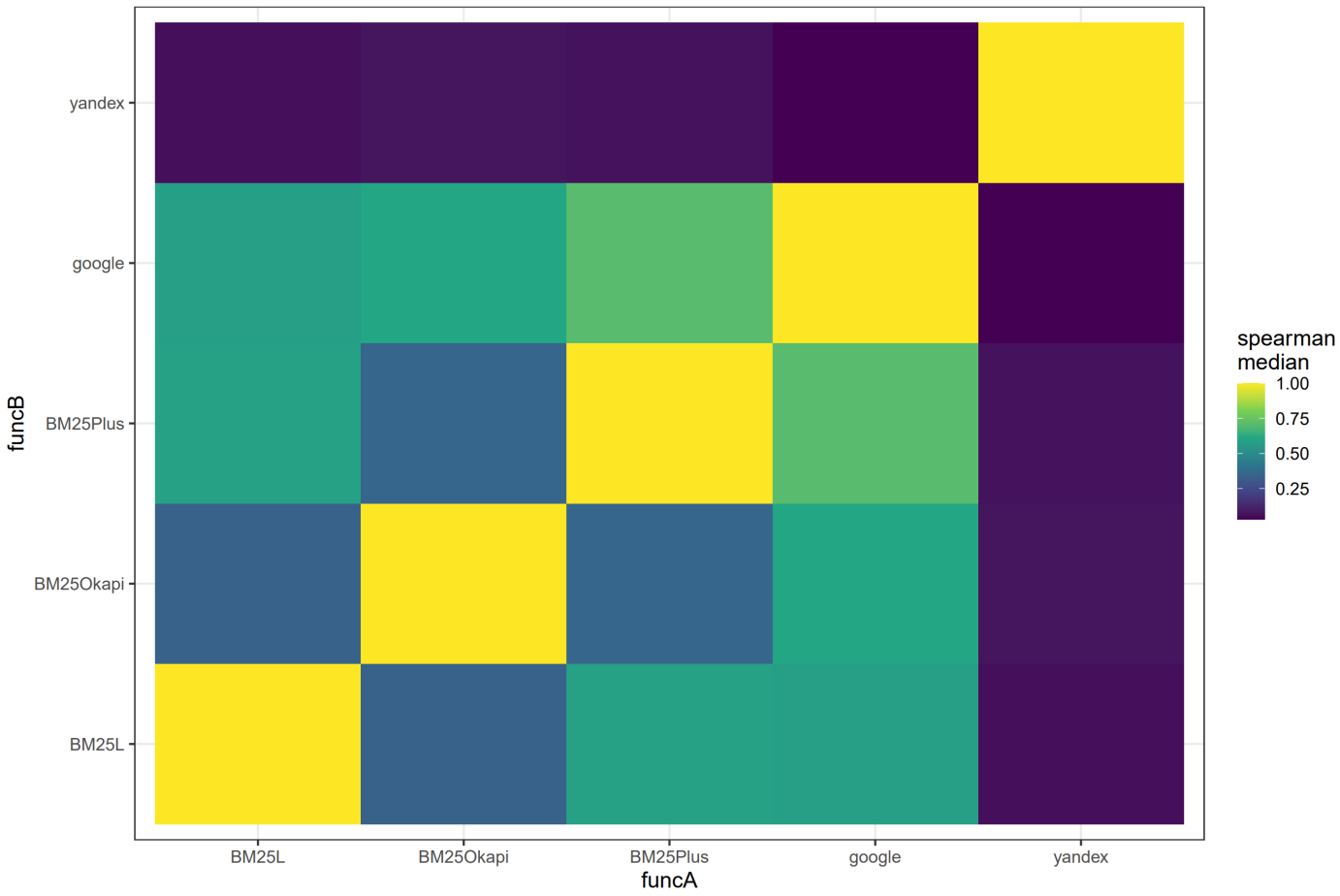


Figure 9 – Median ranking similarity rates by Spearman's coefficient

These disagreements prevent us to make a conclusion that Google really use one of the BM25 functions as a base.

Nevertheless, one thing remains constant – Yandex has a ranking similarity rate close to zero. There are 0.06, 0.07, 0.07 and 0.03 median values matching BM25L, BM25Okapi, BM25Plus and Google's ranking functions. And we are more sure now that it is unlikely Yandex ranks its search results with BM25 function or has some common ranking algorithm's parts with Google.

Conclusions on Chapter 3

As a conclusion on this chapter, we did not find a function having a big similarity rate with engine's ranking. However, it is still a strong fact we learned – neither Google nor Yandex use exactly BM25. And these two engines have completely dissimilar rankings on a certain considered set of documents. Moreover, we developed the system that allows to check if any given ranking function is used by

any given engine. The possible use-case is the following: someone claims it has Google's ranking algorithm, then we are able to implement this algorithm and to check how likely is this usage.

CHAPTER 4. COMPARING ROBOTS' REFRESHINGS OF STATIC AND DYNAMIC PAGES.

The current chapter is an additional for our work. Here we are going to use the other ground truth than in the second chapter. We are going to rely on visitors' user agents. The objective here is to compare how often do robots refresh static pages compare to dynamic one.

4.1. Static pages and dynamic page descriptions.

Actually, we already have static pages in our research – the set of 200 documents, also referred to as chunks. Its original goal is to form the orderings to investigate search engines' rankings. However, we reuse them in this chapter as they are perfectly fit to studying web pages' refreshing – light to be loaded quickly by search robots and never updating.

As a dynamic page, we use a honeypot's index page. It is always updated by the updating script, which provides search robots with unique information each hour. Also, it is easy to produce for robots too.

4.2. Collecting robots' visit dates.

The goal of this chapter is to research all the possible robots' visits, not only the engines' ones. That is why we extract all the visitors from access logs that have the "bot" infix inside a `user_agent` field. Moreover, as we require only the robots that could refresh the pages, it is needed to filter them by the requested page – `uri`. The comprehensive algorithm is presented in the listing 11. There are `index_refresh_dates` standing for honeypot's index page's refreshing dates, and `chunks_refresh_dates` storing refreshing dates corresponding to each chunk.

4.3. Analysing the collected robots' visit dates.

Let us start analysing of the collected dates with the following research question – what robots are most active by the number of refreshing visits. The computations behind are quite simple, we just need to get the number of visit dates

Listing 11 – Finding robots refreshing pages

```
index_refresh_dates = dict() //a key is a bot name and a value is
   its refreshing dates
chunks_refresh_dates = dict() //a key is a chunk index and a value
    is dictionary where a key is a bot name and a value is its
   refreshing dates

for visitor in access_logs:
    uri = visitor["uri"]
    user_agent = visitor["user_agent"]
    date = visitor["date"]

    if user_agent contains "bot":
        bot_name = get_bot_name(user_agent)
        if uri == "/index.html":
            index_refresh_dates.update(key=bot_name, value=date)
        if uri contains "chunk":
            chunk_index = extract_chunk_index(uri) //example:
                "chunks/chunk_1.html" -> 1
            bot_dates = chunks_refresh_dates[chunk_index][bot_name
               ]
            chunks_refresh_dates.update(key=chunk_index, value=
               dict(key=bot_name, value=bot_dates+date)
```

corresponding to each robot. As a result, we observed 11 search bots – AdsBot, AhrefsBot, AmazonBot, BingBot, BLEXBot, CCBot, DuckDuckGoBot, Google-Bot, MauiBot, SemrushBot and YandexBot. The total number of their visits are presented in the figure 10. It is impossible to argue that GoogleBot has the maximum number of refreshing over all bots – about 1200. And it perfectly correlates with the number of detected honeypot's version and the speed of indexing the chunks. The second in our ranking is AhrefsBot, accounting for almost 900 visits. And the last in top 3 is AdsBot with nearly 750 refreshings. Surprisingly, according to these numbers, BingBot is more active than YandexBot, but as we learned from chapter2, Yandex detected much more versions than Bing and what is more – Bing did not index chunk pages at all. The low refreshing number of DuckDuckGo is reasonable too – DuckDuckGo engine uses about 400 sources to produce search results, which could be more active than its own search bot. It should be noted that the dates' range is two months (April 2021 – June 2021) here.

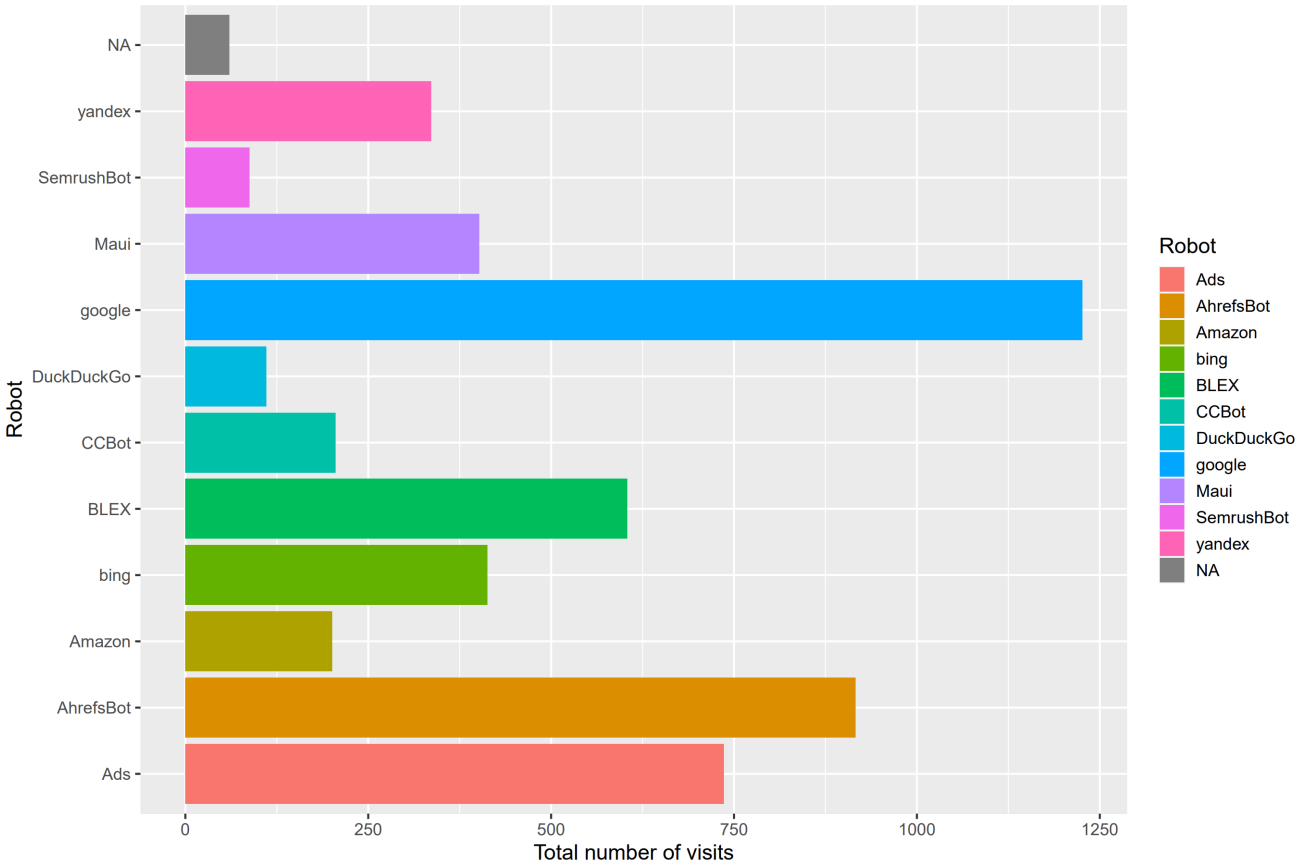


Figure 10 – Example fragment of chunk document

Now we can move to the main question of this chapter – how often do robots refresh dynamic page compared to static ones? To answer this question properly, we build the scatter plot – figure 11. It is the date here on X axe, and chunk indexes on Y axe. As we need to present the honeypot's index page which is a dynamic one, we refer to it as chunk with index -1 for the purpose not to pollute Y axe representation. It is clear at the first sight that the dynamic page is refreshed more frequently. There are only few days when none of the search robots visited the index page. Which could not be said about chunk static pages. Search robots do not often come to re-index a page that does not have a tendency to update its content. And it is quite logical phenomenon since robots are able to adapt.

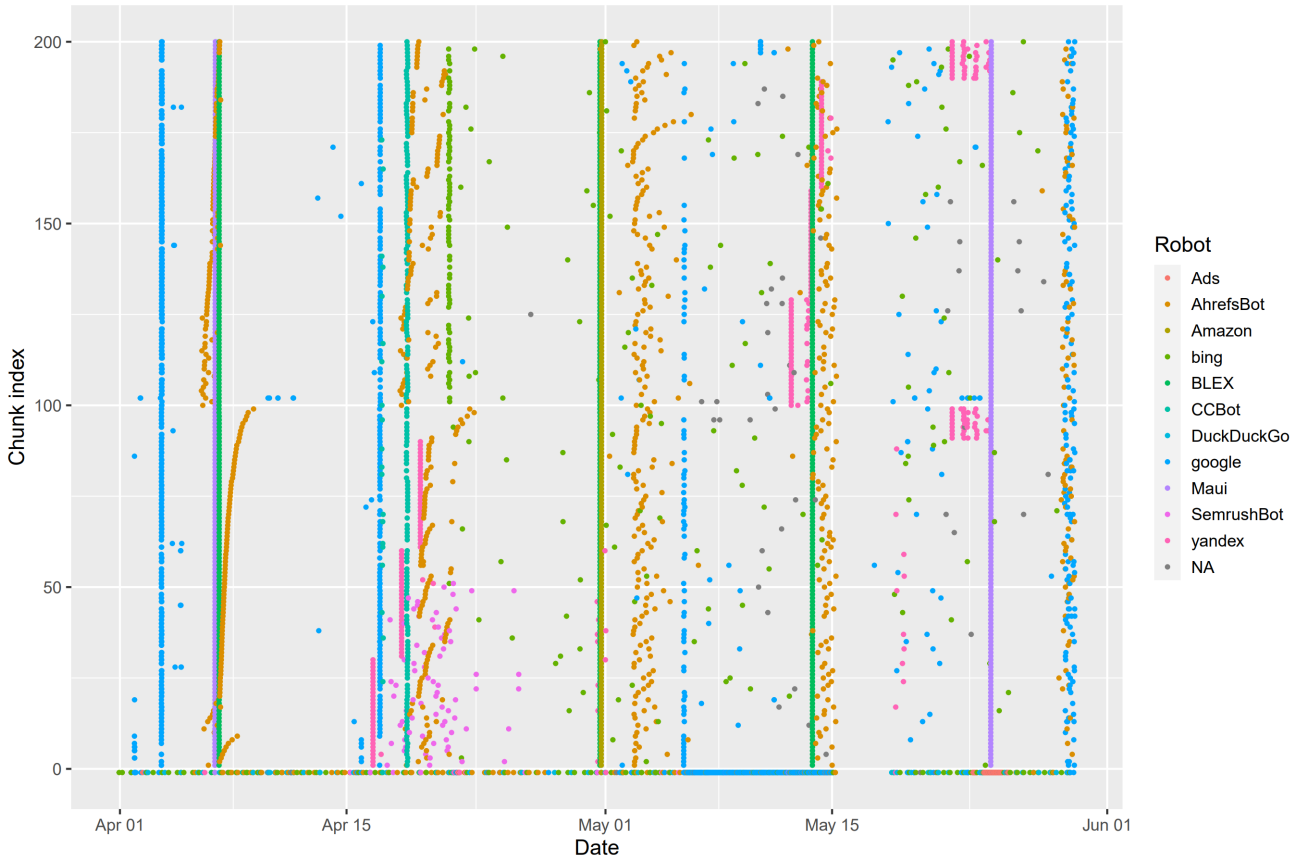


Figure 11 – Example fragment of chunk document

There is one more observation could be made here – different robots have different patterns of refreshing web pages. It is illustrated well with bots' visits of the chunks. However, there are too many robots fused on this plot, and due to the fact we are primarily interested with search engines' bots, we are going to zoom this plot to look at the most valuable bots' patterns.

These bots are BingBot, GoogleBot, YandexBot and AhrefsBot (as it is the most active non-search-engine bot). Their refreshing dates are presented in the figure 12. Here we can see that Ahrefs and Google refresh chunks more systematically than Bing and Yandex, which is confirmed by column-like structure on the scattering plot. Of course, there are some outlying points, but the pattern could be still distinguished. BingBot here seems to try re-indexing the chunks systematically at the first time, but then it was visiting not more than 10-20 pages per day using some specific crawl order. It is still not clear why there are so many visits, and at the same time so small number of observed honeypot's versions. Yandex bot here looks like

the least active. All the visits here happened once we submitted chunks' URLs to the engine with the help of webmaster tools. Without this action, it was a small chance to get the set of documents indexed. As there is a daily limit of submissions that is equal to 30, the indexing corresponding to the 17th, 18th and 19th of April are artificial, as long as the 12th, 13th, 14th of May and about the 21st of May. Consequently, YandexBot itself came to visit the static documents at the 1st of May, the 18th of May and the 22nd-24th of May dates' areas.

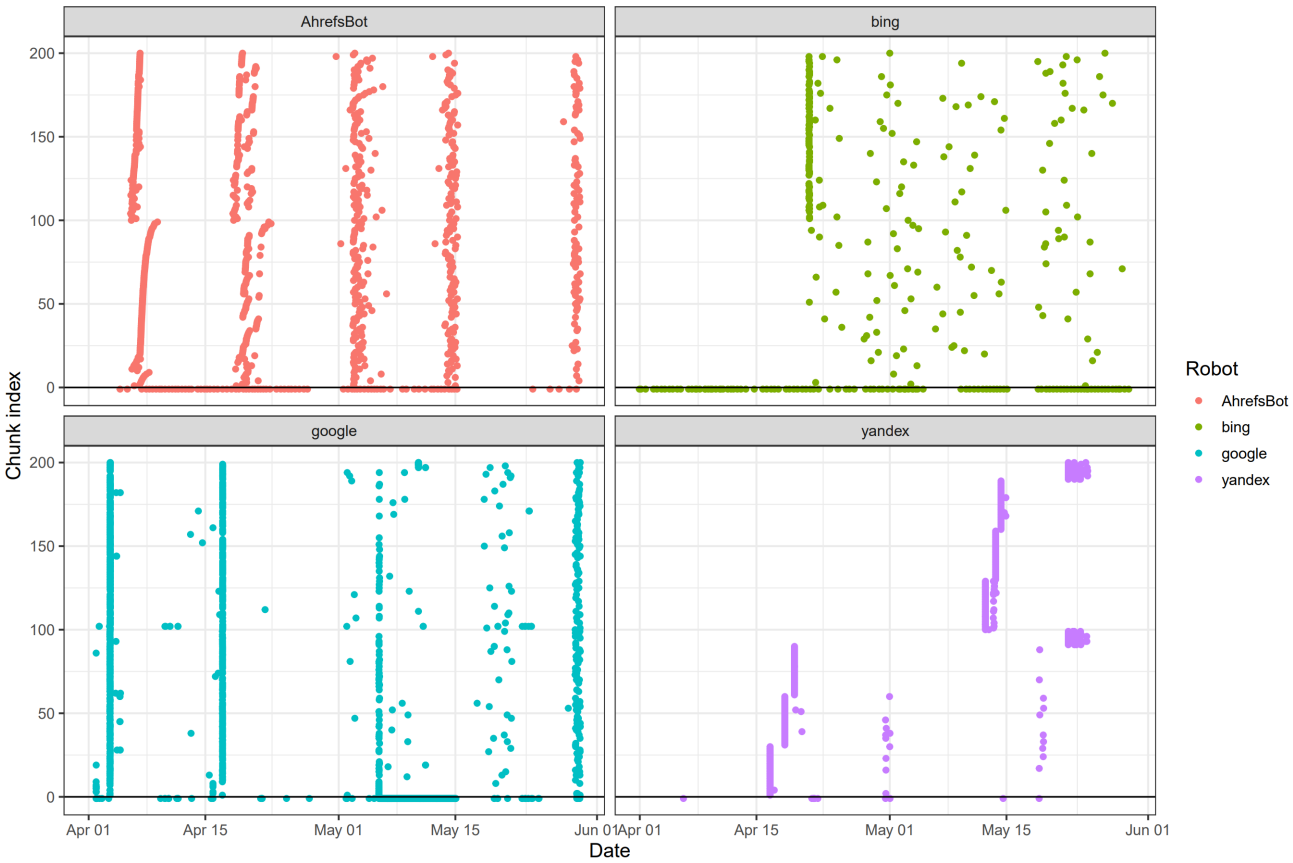


Figure 12 – Example fragment of chunk document

As the previous figure 11, the zoomed plot, figure 12, also confirms the theory that dynamic pages are refreshed more frequently than static ones. There are much more scatters corresponding to the dynamic honeypot's index page.

Conclusions on Chapter 4

In this additional chapter, we presented a method to compare the refreshing of two types of webpages – static and dynamic. We found out that robots tend to

re-index pages with changing content more frequently, than static ones. We also observed some behaviour patterns of robots visits within the limit of our webpages' "dataset".

CONCLUSION

In this research, we created a system to conduct black box audits of search engines. This system is based on honeypots and the Barium Meal test, and it does not rely on search engines' collaboration. This joint ability to control what we provide to a search engine and what gets out of a search engine is, to the best of our knowledge, unique.

We confirmed the system's capacity to infer information by confirming some conclusions with exterior knowledge. We showcased some numerous potential applications of such a tool in a black box auditing context.

We showed an efficient approach to auditing search engines as online black boxes, which are really central pieces of our modern society. We believe such tools have a role to play in allowing societies overview of search engines.

REFERENCES

1 *Trotman A.*, *Puurula A.*, *Burgess B.* Improvements to BM25 and Language Models Examined // Proceedings of the 2014 Australasian Document Computing Symposium. — Melbourne, VIC, Australia : Association for Computing Machinery, 2014. — P. 58–65. — (ADCS '14). — ISBN 9781450330008. — DOI: 10.1145/2682862.2682863. — URL: https://doi.org/10.1145/2682862.2682863.

2 About Swisscows. — 2021. — URL: https://swisscows.com/en/about.

3 Can I add my site to your search results index? – Ask. — 2021. — URL: https://help.ask.com/hc/en-us/articles/360001685614-Can-I-add-my-site-to-your-search-results-index.

4 Canary trap. — 2021. — URL: https://en.wikipedia.org/w/index.php?title=Canary_trap&oldid=1013490013.

5 DuckDuckGo — Privacy, simplified. — 2021. — URL: https://duckduckgo.com.

6 DuckDuckGo vs. Google: An In-Depth Search Engine Comparison. — 2021. — URL: https://www.searchenginejournal.com/google-vs-duckduckgo/301997.

7 European Davids vs Goliath: Privacy search engines aim to challenge Google | Financial Post. — 2018. — URL: https://financialpost.com/technology/european-privacy-search-engines-aim-to-challenge-google.

8 Exalead. — 2020. — URL: https://en.wikipedia.org/w/index.php?title=Exalead&oldid=957282393.

9 geckodriver. — 2021. — URL: https://github.com/mozilla/geckodriver.

10 Google Organic CTR History - Advanced Web Ranking. — 2021. — URL: https://www.advancedwebranking.com/ctrstudy.

11 *Gregersen E.* Bing // Encyclopedia Britannica. — 2021. — URL: https://www.britannica.com/topic/Bing-search-engine.

12 Have it your way with the Mojeek search engine. — 2012. — URL: https://web.archive.org/web/20130526051237/http://www.pandia.com/sew/173-have-it-your-way-with-the-mojeek-search-engine.html.

13 How can I remove my own listing from Lycos Search results? : Lycos. — 2017. — URL: https://helpdesk.lycos.com/support/solutions/articles/29000011678-how-can-i-remove-my-own-listing-from-lycos-search-results.

14 Okapi BM25. — 2021. — URL: https://en.wikipedia.org/w/index.php?title=Okapi_BM25&oldid=1008742667.

15 Personalised Search. — 2021. — URL: https://yandex.com/company/technologies/personalised_search.

16 Question answering. — 2021. — URL: https://en.wikipedia.org/w/index.php?title=Question_answering&oldid=1022819151.

17 rank-bm25 · PyPI. — 2020. — URL: https://pypi.org/project/rank-bm25.

18 Search Engine Market Share Russian Federation. — 05/2021. — URL: https://gs.statcounter.com/search-engine-market-share/all/russian-federation.

19 Search Engine Market Share Worldwide. — 05/2021. — URL: https://gs.statcounter.com/search-engine-market-share.

20 selenium · PyPI. — 2018. — URL: https://pypi.org/project/selenium/.

21 SEO Starter Guide: The Basics | Google Search Central. — 2021. — URL: https://developers.google.com/search/docs/beginner/seo-starter-guide.

22 Spearman's rank correlation coefficient. — 2021. — URL: https://en.wikipedia.org/w/index.php?title=Spearman%5C%27s_rank_correlation_coefficient&oldid=1023950377.

23 Startpage.com - The world's most private search engine. — 2021. — URL: https://web.archive.org/web/20210401002959/https://www.startpage.com/.

24 The top 500 sites on the web. — 2021. — URL: https://www.alexa.com/topsites.

25 Top 50 Search Engines. — 2020. — URL: https://www.marketingcourses.com/top-50-search-engines.

APPENDIX A. DESCRIPTIVE TABLE OF SEARCH ENGINES.

Table A.1 – Search engines description

Name	Launch Year	Number of Supported Languages	Market Share (in percent)[1]	Short Description
Google	1997	149	92.18	The most visited website in the world [24].
Startpage	1998	82	< 0.1	Pays Google to use its search results, but removes all trackers and logs to reach high privacy. No ads, no data mining [23].
Bing	2009	40	2.27	Launched by Microsoft. "Decision engine" that displays more retrieved information than typical [11].
DuckDuckGo	2008	> 70	0.59	Emphasizes privacy protection, avoids filter bubble and personalized search results [5]. Gets results from over 400 sources [6].
Ask.com	1996	1	< 0.1	Originally based on Question Answering [16].
Mojeek	2004	3	< 0.1	Crawler-based, provides independent search results using its own index [12].
Exalead	2000	2	< 0.1	Apply semantic processing and faceted navigation to Web data volumes and usage [8].
Lycos	1995	12	< 0.1	Was one of the first available on the internet [25].
Yandex	1997	8	0.69	Almost as popular as Google in Russian Federation [18]. Personalized search results [15].
Swisscows	2014	9	< 0.1	Respect privacy, family-friendly content [2].

[1]The percent is taken from Search Engine Market Share Worldwide [19]

APPENDIX B. EXAMPLE FRAGMENTS OF FUSION LOGS

Listing B.1 – Fragment of Google fusion log

```
{
    'engine': 'google',
    'version': 254,
    'visitors': [
        {
            'remote_addr': '91.201.244.224',
            'remote_port': '38661',
            'user_agent': 'Mozilla/5.0 (Macintosh; Intel Mac OS X
              10_11_6 AppleWebKit/601.7.7(KHTML, like Gecko)
              Version/9.1.2 Safari/601.7.7',
            'uri': '/index.nginx-debian.html',
            'time': '13/03/2021 02:58:38'
        },
        ...
        {
            'remote_addr': '66.249.75.177',
            'remote_port': '44034',
            'user_agent': 'Mozilla/5.0 (compatible; Googlebot
              /2.1;+http://www.google.com/bot.html)',
            'uri': '/robots.txt',
            'time': '13/03/2021 03:36:14'
        },
        {
            'remote_addr': '66.249.75.175',
            'remote_port': '56082',
            'user_agent': 'Mozilla/5.0 (Linux; Android 6.0.1;
              Nexus 5X Build/MMB29P) AppleWebKit/537.3(KHTML,
              like Gecko) Chrome/89.0.4389.89 Mobile Safari
              /537.36 (compatible; Googlebot/2.1+http://www.
              google.com/bot.html)',
            'uri': '/index.html',
            'time': '13/03/2021 03:36:14'
        }
       ...
}
```

Listing B.2 – Fragment of Yandex fusion log

```
{
    'engine': 'yandex',
    'version': 471,
    'visitors': [
        {
            'remote_addr': '114.119.147.86',
            'remote_port': '38644',
            'user_agent': '(compatible;PetalBot;+https://aspiegel.
              com/petalbot)',
            'uri': '/robots.txt',
            'time': '22/03/2021 04:32:10'
        },
        ...
        {
            'remote_addr': '5.45.207.125',
            'remote_port': '60684',
            'user_agent': 'Mozilla/5.0 (compatible; YandexBot/3.0;
               +http://yandex.com/bots)',
            'uri': '/robots.txt',
            'time': '22/03/2021 04:34:28'
        },
        {
            'remote_addr': '5.45.207.125',
            'remote_port': '60684',
            'user_agent': 'Mozilla/5.0 (compatible; YandexBot/3.0;
               +http://yandex.com/bots)',
            'uri': '/index.html',
            'time': '22/03/2021 04:34:32'
        }
       ...
}
```

APPENDIX C. ADDITIONAL PLOTS BASED ON COLLECTED ROBOTS RECORDS

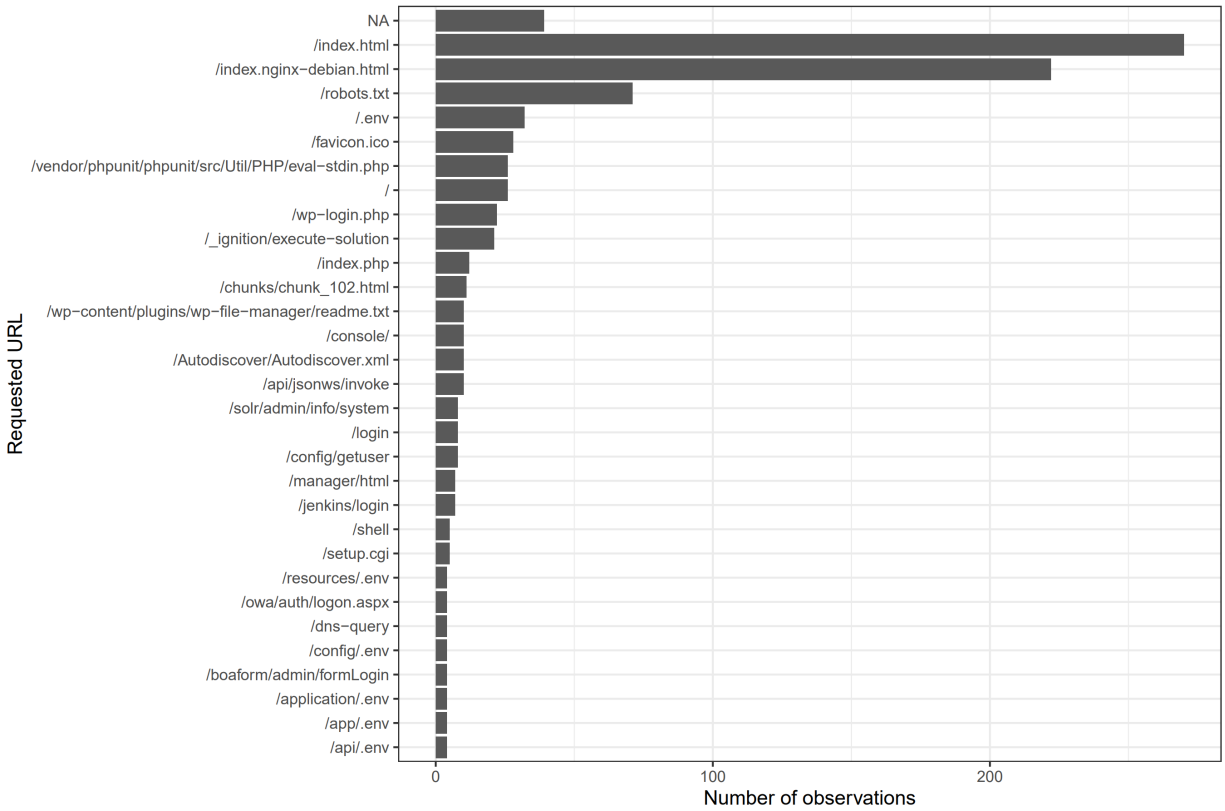


Figure C.1 – All URL requests of potential robots

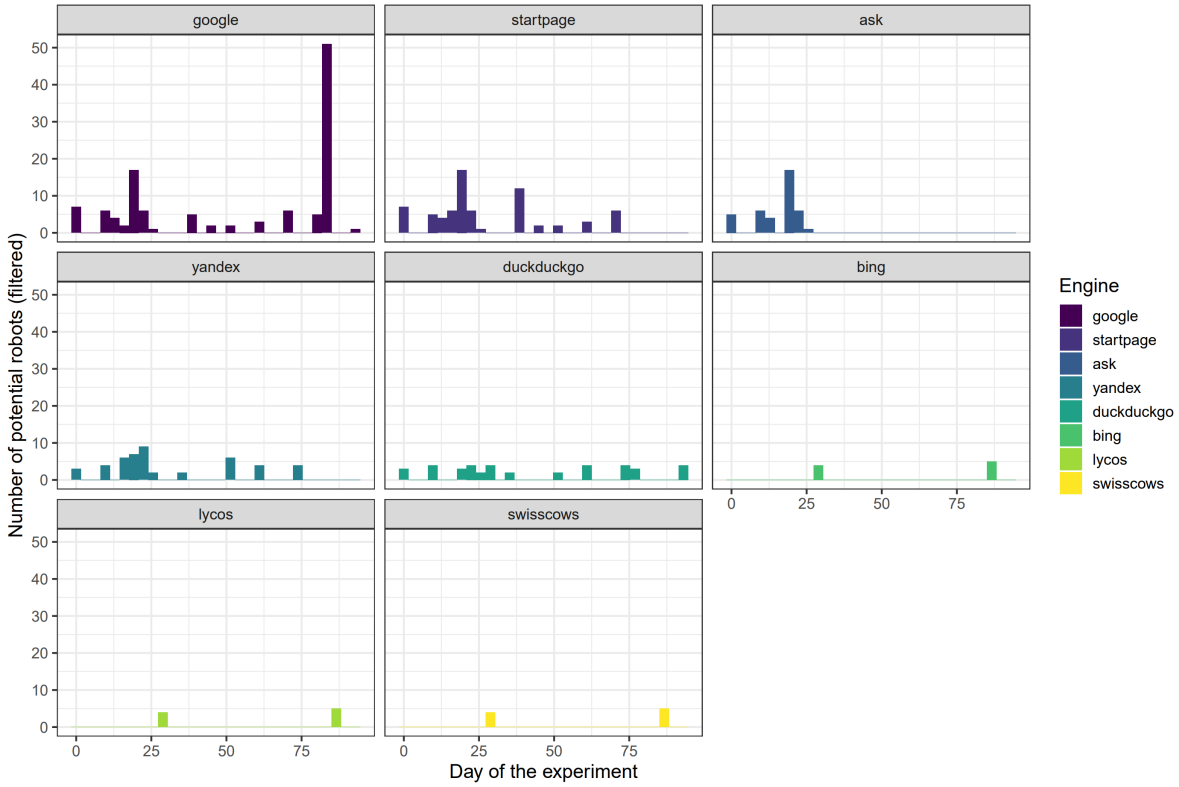


Figure C.2 – Number of potential robots by day of the experiment

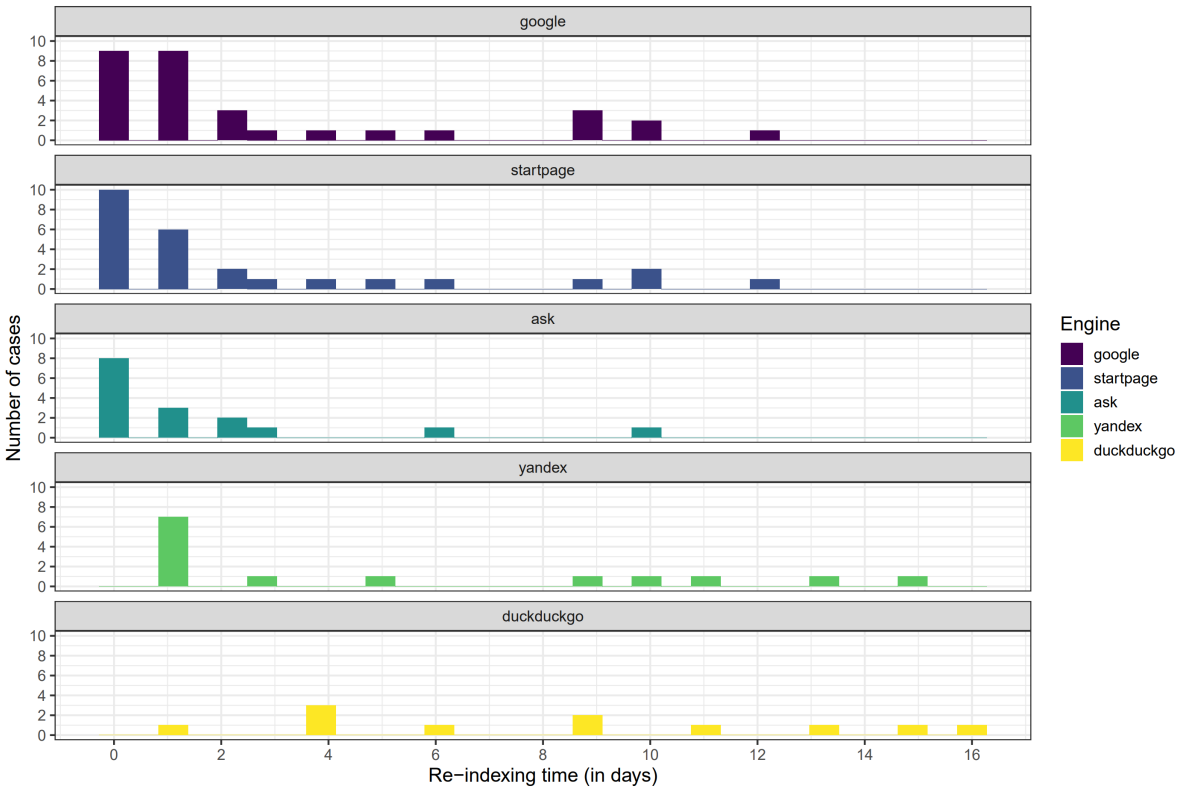


Figure C.3 – Re-indexing time (in days) for each search engine