# Search System for planning your spare time.

Nadezhda Bugakova
Saint-Petersburg Academic University
Russia, Saint-Petersburg
bugakova.nadya@gmail.com

Elizaveta Tretiakova
Saint-Petersburg Academic University
Russia, Saint-Petersburg

## ABSTRACT

This paper provides a full description of Information Retrieval project aiming to search different types of activities for user's spare time. It aggregates theater plays, different performances, master classes, cinema offers and many other events taking place in the city. System allows user to search activities near a chosen place. Also the system is updating content so as not to show out of date and irrelevant information.

## KEYWORDS

IR, Information Retrieval

**ACM Reference Format:**
Nadezhda Bugakova and Elizaveta Tretiakova. 2017. Search System for planning your spare time.. In *Proceedings of ACM Conference (Conference'17)*, Jennifer B. Sartor, Theo D'Hondt, and Wolfgang De Meuter (Eds.). ACM, New York, NY, USA, Article 4, 4 pages. https://doi.org/10.475/123_4

## 1 INTRODUCTION

Information Retrieval consists of the parts you can see on Fig. 1. Each search system contains all these parts (or analogues of some kind). This paper sheds light on all stages of the system architecture but query processing part which was implemented using standard libraries. Paper also provides a description of each step, decisions we made during development and our reasons to make these ones and not others. Full sources can be found in our repository [1].

In Sec. 2 we described the basic part of any search system – data acquisition. We showed the architecture of system in Sec. 2.1. Politeness of crawler and issues are described in sections 2.2 and 2.3.

In Sec. 3 data processing and indexing are described. Data processing description is in 3.1 and methods of indexing and storages description are placed in Sec. 3.3. The problems and solutions are provided in Sec. 3.4.

## 2 DATA ACQUISITION

### 2.1 Architecture

The main class Crawler runs the whole process. It is also responsible for exploring new sites and pages, which are then stored as instances of the corresponding classes (which also provide politeness checks) in a specially designed queue. The necessity for creating our own

[1]https://github.com/N-buga/spbau-IR2017

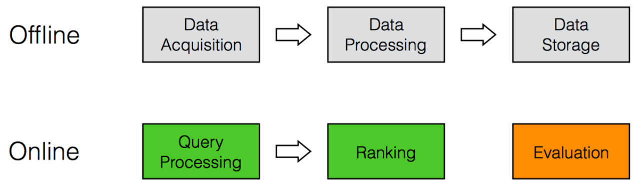


**Рис. 1: High-level architecture.**

queue grew from the need to update our data. So we decided that at the first iteration, while we cannot know for sure the exact update policy we need, the simple restriction on the number of sites we are looking at combined with the cycled queue should work for us (Crawler will get the old sites from time to time and so the update will be performed "automatically"). In other words, we implemented cycled queue together with restriction on its size so that the crawler would return to the start of queue with old pages after some time.

To explore internet space crawler needs to have a start point from which it can begin. We defined such a start point as an initial configuration of the url queue. Than means that when crawler starts gathering sites queue has to contain some initial urls already. We are putting two url in the queue. They are 'https://afisha.yandex.ru/' and 'https://kudago.com'. We were considered this sites to be very relevant to our sphere of search. We managed to crawl more than 50000 pages int he end.

Apart from the data gathering we also implemented an automated "healthcare"system of checkpoints which periodically backs up a state of the system just in case something goes wrong. It helps our system to crawl from the point it was stopped by exception or manually.

### 2.2 Politeness

First of all, we name ourselves ending with "spider". We also search each site's robots.txt file and get the NOFOLLOW, NOINDEX, NOARCHIVE flags (and obey the corresponding prescriptions) as well as a crawl-delay if present. The default crawl-delay in our system is 1 second. We came to this empirically-estimated constant after we had encountered some problems with other approaches. For example, if we waited 10 times of the page loading, we could be stuck for quite a while in case something happened and thus prolonged the download time (e.g. internet connection loss on closing of a laptop and restore when we are back after several hours; now you can multiply this by 10 and see our reasons to pick a constant).

## 2.3 Problems&Solutions

First problem, already mentioned above, was about politeness. At first we used to wait 10 times more than the time it took us to download the document. The approach seemed legit at first, however it showed to be extremely inefficient. One of the causes was an unstable internet connection. For example, we had to first wait for a long time just to download a document, only to then wait ten times this value. To solve the problem we decided to change the delay in a way it is now.

Despite these changes crawling still has a problem with loading large documents – it takes time and we cannot help here. This problem also led to another one. During the crawling different urls are encountered by crawler. Some of them may be incorrect or not supported by our crawler. Therefore crawler often fell down with different unexpected errors. We had to handle errors each time they were raised.

Unexpected errors (such us ChunkedEncodingError or MissingSchema) also have another issue. Since crawler works very long we couldn't look after it all the time. It led to a waste of resources and waste of all data we've already loaded because we had to rerun the whole system on errors raising. To solve this problem checkpoints were implemented. Once in 1000 steps all instances are saved to a file. If the crawler stops due to some error we can load all instances again and all urls it has previously downloaded would be in the url queue. That means it doesn't forget the work already done.

There was one more small issue related to storing documents. In the beginning we used to save them in files with names containing the urls. It turned out that some urls can be so large that the file system refuses to save files with such long names. So we switched from using urls to using their hashes instead and added a description file where relations between urls and hashes are stored.

## 3 DATA PROCESSING AND STORAGE

Once data is acquired we need to process it and store in a way that lets us handle user requests effectively. One of the widely used solutions is to create an index on a preprocessed text.

### 3.1 Data extraction and text processing

Two major logical parts were implemented: a rough page processing, with stemming and stop-words removal, which was then used to calculate all the necessary statistics and entity extraction, where we tried to extract some specific information and be as precise as possible. For two sites with a well-organised structure (afisha.yandex.ru, www.mariinsky.ru), we implemented separate entity retrievers (developers of the latter site were quite creative in their approach to the architecture: most of the information about the event is easily extracted from the url itself, which is quite convenient), for all the others some general heuristics were applied.

Giving some more detail, we used the awesome Python NLTK library: SnowballStemmer did a nice job on stemming (the possibility to set the language we need was a good bonus since we process sites mostly in Russian and not in English), the NLTK's corpus of stop-words (also in several languages) was used for clearing the text.

### 3.2 Entity Extraction

For us, the most effective way to extract features was a simple manual parsing of pages with a known structure (as we did for *afisha* and *mariinski*). However, having lots of other sites, we had to come up with some idea for them either. So here the combination of the BeautifulSoup and dateutil.parser libraries did the trick (with a certain level of quality): the former is used to extract all the possibly meaningful text from the page, and the latter is used particularly for date-time extraction. Although this approach has its problems, e.g. it extracts all the dates from the page, and we will have to work out the way to smartly pick the proper date (the actual date of the show) in the future.

Apart from the date, we also try to obtain the name in the default mode. Other features are still not available. At the same time for the known sites the range of extracted entities is, quite obviously, wider: the type an event, the venue, the price and the city.

### 3.3 Indexing

We used two types of indexing: page attribute file and inverted index.

Page attribute file represents bare page statistics. Usually it may be link or word count, or a probability of the page to be spam. In our system we decided to use only link count, total word count and unique word count, which together allow us to emphacise the page quality properly. We store page attribute file in a PostreSQL database (and wrote a special wrapper for this purpose). In future we are going to store more information in the database; we will get into more detail on this in the next section.

Inverted index is the structure in which each word (or phrase) is matched with some information from document about this word (or, respectively, phrase). For example, this information can be identificatots of the documents that contains the word or the positions of the word in the documents, or count of these words in the documents.

We store ids of documents, positions and count in our inverted index. As pure python implementation was very slow we rewrote it using Spark to make it faster. Whole index is stored in a binary file but the system keeps metainformation in memory. For each word metainformation contains number of bytes to skip from the beginning of the binary file to get to the part of index where information about the word are stored and number of bytes this information occupy. So when we want to get information about word from index there's no need to load whole index to memory. We use metainformation to load just that many bytes as we need. After that they just need to be deserialised.

To store this structure we use a plain text file.

Unlike the page attribute file, this type of index may have issues with memory and performance. We focused on the memory and implemented two-pass indexing ( 2). It does two document traversals: the first one, to estimate the per-word space required, and the second one,to actually store the data. The first traversal eliminates the need in keeping the index in memory. Instead, we can write it just-in-time.

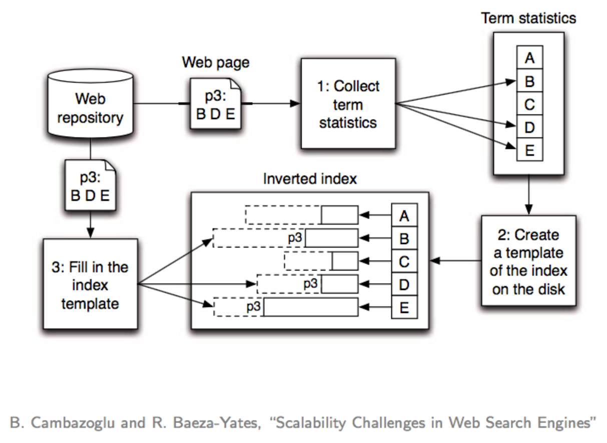

Term statistics
Web page
Web repository
p3: B D E
1: Collect term statistics
A
B
C
D
E
p3: B D E
Inverted index
3: Fill in the index template
p3
p3
p3
2: Create a template of the index on the disk
B. Cambazoglu and R. Baeza-Yates, "Scalability Challenges in Web Search Engines"

**Рис. 2: Two-pass index.**

## 3.4 Problems&Solutions

On this stage we encountered a serious problem with serialization. In the beginning of the project we used to save the state of the system using standard python serialization protocol called pickle. When we added text processing and indexes, our classes changed significantly, so that the older versions were completely incompatible with the newer ones. So we implemented our own serialization and deserialization in addition to the existing pickle.

## 4 SEARCH AND FEATURES

### 4.1 Search

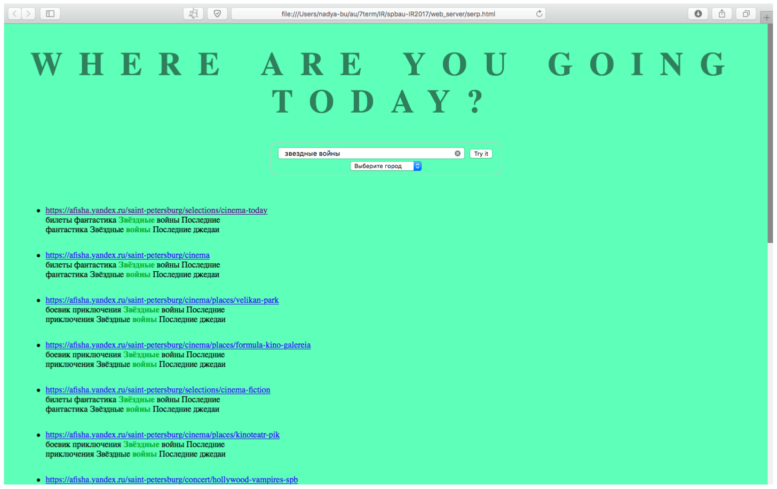
To rank documents we decided to use BM25 metric of relevance. We calculated it after getting relevant documents.

$$BM25_d = \sum_{t \in q} \log \left( \frac{N}{df(t)} \right) \frac{(k1+1)tf(t,d)}{k1\left((1-b) + b\frac{dl(d}{dl_{ave}}\right) + tf(t,d)}$$

It is a parametric model so it requires to set two constants: k1 and b. As was shown, for most of the cases $k_1 \in [1.2, 2]$ and $b = 0.75$ work quite well, so $k1$ was set to 1.5 and $b$ was set to 0.75. We used the information stored in the inverted index and page attribute file to calculate the ranking.

### 4.2 Features

As mentioned above, our main feature is localization. We described how exactly different entities are extracted. There is still quite a room for development but for the time being we implemented only localization, i.e. we made use of the city attribute only. Actually, it was achieved quite simply: we showed documents from which we managed to extract the city and this city matched the user query. Alternatively, we could just increase the metric for such documents. This way user would more likely see relevant documents in a wrong city rather than irrelevant documents in the chosen one. We decided to proceed with the first option, however, the second one would probably show better results. So we are going to compare both options on the evaluation stage and then we'll see which one should be used.


file:///Users/nadya-bu/au/7term/IR/spbau-IR2017/web_server/serp.html
WHERE ARE YOU GOING TODAY?
звездные войны
Try it
Выберите город

- https://afisha.yandex.ru/saint-petersburg/selections/cinema-today
  билеты фантастика **Звёздные** войны Последние
  фантастика Звёздные **войны** Последние джедаи
- https://afisha.yandex.ru/saint-petersburg/cinema
  билеты фантастика **Звёздные** войны Последние
  фантастика Звёздные **войны** Последние джедаи
- https://afisha.yandex.ru/saint-petersburg/cinema/places/velikan-park
  боевик приключения **Звёздные** войны Последние
  приключения Звёздные **войны** Последние джедаи
- https://afisha.yandex.ru/saint-petersburg/cinema/places/formula-kino-galereia
  боевик приключения **Звёздные** войны Последние
  приключения Звёздные **войны** Последние джедаи
- https://afisha.yandex.ru/saint-petersburg/selections/cinema-fiction
  билеты фантастика **Звёздные** войны Последние
  фантастика Звёздные **войны** Последние джедаи
- https://afisha.yandex.ru/saint-petersburg/cinema/places/kinoteatr-pik
  боевик приключения **Звёздные** войны Последние
  приключения Звёздные **войны** Последние джедаи
- https://afisha.yandex.ru/saint-petersburg/concert/hollywood-vampires-spb

**Рис. 3: Interface example.**

### 4.3 Problems&Solutions

It turned out that the chosen way to extract text was not the best one. We had extracted content from aside elements such as forms, menu items or recommendations. Consequently, for example, if a query is about something popular, recommendations could mess up the ranking. The problem was solved by checking more conditions during the extraction. Before this problem was discovered, we had been taken the text from the particular html elements ('div', 'span' and 'body'). Now we added restrictions on the elements, content from which we aren't consider anymore (i.e 'aside', 'meta' and some others).

## 5 INTERFACE

Forf the search system it is quite naturally to implement web-interface for it. Search page has an animated title, a search box in the middle and a dropdown list for city selection. Each point of the search output also has a snippet with query words highlighted, so the user could understand the general idea of the page. The map is under the search output. It draws your location on the map. As a development, the events locations will be shown as well. Fig. 3 shows the interface.

### 5.1 Architecture

One of the issues with using a web interface is the interaction between the Python backend and the JavaScript frontend. So we addressed the issue by using a server-client architecture. That means that our search system can be turned into a web-service easily. For the server side we use an asynchronous framework called Asyncio.

As was mentioned above, JavaScript was used for interface. CSS framework helps us to keep all the elements in the same style. Also we used Animejs for text animation. This framework offers a variety of customizable effects.

If the user does not pick any city, we try to locate him using the browser's 'navigation' object; if this does not help as well, we use the default city - Saint Petersburg.

In order to save memory, we do not store the page text, neither partly nor the whole. Thus, in order to present some snippets to the user, we download the page again each time we want to show it in the search results. Then we extract sample occurences of the words from the query in the text and present them to the user (highlighted).

### 5.2 Problems&Solutions

Due to the described implementation of the snippets, we experience some troubles when a site decides to block us (in our case, it was yandex.ru). This block results into our inability to download the page text, so we do not provide a snippet as well (though the other snippets and the link are still presented, of course).

Another problem is the speed. As the number of documents in our base grows, the indexes grow, and thus the whole query processing takes more time, so the response delay becomes too noticeable. We are working on this bug.

## 6 EVALUATION

To evaluate our system, we created two sets of tasks: the first one was designed to get a broad, qualitative feedback about the system in general (e.g. user experience, visual attractiveness) along with the relevance feedback for a number of queries (the user was given a number of scenarios which required him or her to do come up with a query for our system and try to find the relevant links from the ones provided by our system). This task was completed by 3 assessors.

The other set was designed to obtain purely quantitative feedback and consisted of several queries with the system's responses, which were to be ranked by our assessors. 5 assessors completed this task (one person participated in both evaluations).

Basing on the feedback, we identified several ways the UX of our system can be improved. These include a better page layout, fixing latency issues and, maybe as a future work, dealing with word relations and spelling (so that, for example, the query "кино кроме звёздных войн"would actually return the pages without the words "звёздных войн and not vice versa as it is now).

For the quantitative offline evaluation the following popular metrics were chosen: P@5, Reciprocal Rank, AP and DCG. We are not presenting the results for each user separately. Instead, we present the mean value. The queries are shown in Table 1.

| | |
|---|---|
| query1 | звёздные войны расписание завтра |
| query2 | куда сходить спб |
| query3 | эрмитаж экспозиции |
| query4 | расписание кино спб |
| query5 | корейское кафе |

**Таблица 1: Queries**

We asked 5 assessors to estimate the search output for each query in the scale from 0 to 4 where 0 means not relevant result and 4 means relevant and useful. We considered links with the grade 2 or more to be relevant.

The result of the qualitative evaluation may be found in the Table 2.

| | mean P@5 | mean RR | mean AP | mean DCG |
|---|---|---|---|---|
| query1 | 0.64 | 0.4 | 0.52 | 13.5 |
| query2 | 0.64 | 1 | 0.85 | 25.56 |
| query3 | 0.52 | 0.5 | 0.572 | 11.7 |
| query4 | 0 | 0 | 0 | 6.048 |
| query5 | 0.2 | 0.6 | 0.43 | 9.21 |

**Таблица 2: Evaluation results**

We would like to thank the QueueUnderflow and Finding Friends teams for the thorough and extensive help they provided.

## 7 CONCLUSION

There is a number of things that are to be done in the future. That is, for example, speed. Our system is quite slow and user has to wait a long time for the results while we are handling the query. It can be fixed via rewriting some parts in other faster languages or parallelizing system.

Another thing that would be nice to have is extracting other entities and using them for more accurate location detection and personalization. Even though we spent a lot of time on city extracting it may still be improved.

Events representation is also offers a number of ways of development. First of all, their locations may be displayed on a map along with short descriptions. It would help users to identify the most conveniently located events and read information about them at a glance. At the same time, snippets might be extended with characterizing pictures of the events, which would also ease the consumption of the information we give to the user.

One more opportunity to improve the system is setting a certain site quality level in order to favour good-looking well-organised sites rather then link farms or trash sites. That would hopefully influence the user trust to our system in a positive way.

Despite all above issues, the proof of concept might be considered successfull, and the experience of implementing a complex, multicomponental system from scratch (from design to user interfaces and evaluation) is without concern invaluable.