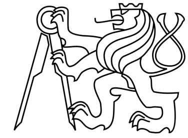
Czech Technical University in Prague
Faculty of Information Technology



ASSIGNMENT OF BACHELOR’S THESIS

Title: Search and filtering system for HelpStone.org
Student: Jind ich Pila
Supervisor: Ing. Vít Steklý
Study Programme: Informatics
Study Branch: Web and Software Engineering
Department: Department of Software Engineering
Validity: Until the end of summer semester 2018/19

Instructions

Improve search and filtering non-profit projects on global crowdsourcing web platform HelpStone.org. The current implementation of the search engine is inadequate. It does not use all existing metadata from our catalogue of non-profits projects and does not deal properly with linguistics. Also, do not forget to evaluate the relevance of the results found.

- Review technologies used by HelpStone.org.
- Review current data storage and existing data.
- Review current search and filtering implementation.
- Research existing search and filtering practices and tools in context of HelpStone project.
- Implement search and filtering system and expose it via an API. Leverage existing open source or cloud solutions when appropriate.
- Test your solution.

References

Will be provided by the supervisor.

Ing. Michal Valenta, Ph.D.
Head of Department

doc. RNDr. Ing. Marcel Ji ina, Ph.D.
Dean

Prague November 14, 2017

FACULTY
OF INFORMATION
TECHNOLOGY
CTU IN PRAGUE

Bachelor's thesis

Search and filtering system for HelpStone.org

Jindřich Pilař

Department of Software Engineering
Supervisor: Ing. Vít Steklý

2018-05-15

Declaration

I hereby declare that the presented thesis is my own work and that I have cited all sources of information in accordance with the Guideline for adhering to ethical principles when elaborating an academic final thesis.

I acknowledge that my thesis is subject to the rights and obligations stipulated by the Act No. 121/2000 Coll., the Copyright Act, as amended, in particular that the Czech Technical University in Prague has the right to conclude a license agreement on the utilization of this thesis as school work under the provisions of Article 60(1) of the Act.

In Prague on 2018-05-15 . . . .

Czech Technical University in Prague
Faculty of Information Technology
© 2018 Jindřich Pilař. All rights reserved.
This thesis is school work as defined by Copyright Act of the Czech Republic. It has been submitted at Czech Technical University in Prague, Faculty of Information Technology. The thesis is protected by the Copyright Act and its usage without author's permission is prohibited (with exceptions defined by the Copyright Act).

Citation of this thesis

Pilař, Jindřich. *Search and filtering system for HelpStone.org.* Bachelor's thesis. Czech Technical University in Prague, Faculty of Information Technology, 2018.

Abstrakt

Práce popisuje implementaci vyhledávání neziskových, vícejazyčných projektů na mezinárodní platformě HelpStone.org. Hlavním tématem je vyhledávání ve strukturovaných a textových datech, které mají varianty v mnoha, potencionálně všech, jazycích. Vyhledávání zvládá práci s afixy, překlepy, synonymy a zvládá full-textové vyhledávání v metadatech jako jsou kategorie nebo země. Jádrem vyhledávání je open source technologie Elasticsearch, distribuovaný vyhledávací nástroj založený na technologii Lucene. Dalšími použitými technologiemi jsou PHP, Symfony Framework a Doctrine ORM.

Klíčová slova Elasticsearch, vícejazyčné vyhledávání, full-textové vyhledávání, PHP

Abstract

This thesis describes implementation of search on international crowdfunding platform HelpStone.org. The main issue being addressed is searching among structured and text data with translations in many, potentially all, languages. This search is able to handle different forms of the same word, typos, synonyms and full-text search in metadata like categories or countries. The core of the solution is an open source technology Elasticsearch, distributed search engine based on Lucene. Other technologies used are PHP, Symfony Framework and Doctrine ORM.

Keywords Elasticsearch, multilingual search, full-text search, PHP

Contents

Introduction **1**

1 Goals **3**

2 HelpStone **5**

2.1 How HelpStone works . . . 5
2.2 Software . . . 6
2.2.1 Docker . . . 6
2.2.2 MySQL . . . 6
2.2.3 PHP . . . 7
2.2.4 Symfony . . . 7
2.2.5 Doctrine 2 ORM . . . 7
2.2.6 RabbitMQ . . . 7
2.2.7 Redis . . . 8
2.3 Data model . . . 8
2.3.1 Translations . . . 8
2.3.2 Metadata . . . 8
2.3.3 Text fields . . . 10
2.3.4 Timeline . . . 10
2.3.5 Size . . . 10
2.4 Current search . . . 11
2.4.1 Features . . . 12
2.4.2 Missing features . . . 12
2.4.3 Implementation . . . 12
2.5 Requirements . . . 12
2.5.1 Functional requirements . . . 12
2.5.1.1 User interface . . . 12
2.5.1.2 Stone . . . 12
2.5.1.3 Code . . . 13

2.5.1.4 Date 13
2.5.1.5 GPS coordinates 13
2.5.1.6 Filtering 13
2.5.1.7 Languages 13
2.5.1.8 Synonyms 14
2.5.1.9 Typos 14
2.5.1.10 Diacritics 14
2.5.2 Non-functional requirements 14
2.5.2.1 Sensitive data 14
2.5.2.2 Technology compability 14
2.5.3 Data size 15
2.5.3.1 Maintainability 15
2.6 Conclusion 15

3 Existing technologies 17
3.1 MySQL 17
3.2 Lucene Core 17
3.3 Elasticsearch 18
3.3.1 Maturity 19
3.4 Solr 19
3.4.1 Maturity 20
3.5 Conclusion 20

4 Elasticsearch theory 21
4.1 Inverted index 21
4.2 Relevance 21
4.2.1 Recall 21
4.2.2 Precision 22
4.2.3 Recall and Precision relation 22
4.3 Ranking and Scoring 22
4.4 Term frequency 22
4.5 Cluster 23
4.6 Data types 23
4.6.1 Text 23
4.6.2 Keyword 24
4.6.3 Numeric 24
4.6.4 Boolean 24
4.6.5 Date 24
4.6.6 Geo-point 24
4.6.7 Object 24
4.6.8 Array 24
4.6.9 Nested 24
4.7 Analysis 25
4.7.1 Character filters 25

4.7.1.1 HTML filter 25
4.7.2 Tokenizer 25
4.7.3 Token filters 25
4.7.3.1 Lowercase filter 25
4.7.3.2 ASCII folding filter 25
4.7.3.3 Stemmer filter 27
4.7.3.4 Hunspell filter 27
4.7.3.5 Keyword repeat filter 27
4.7.3.6 Unique filter 27
4.7.3.7 Synonym filter 27
4.7.3.8 Stop filter 28
4.8 Queries 28
4.8.1 Text queries 28
4.8.1.1 Match Query 28
4.8.1.2 Phrase Query 28
4.8.2 Term queries 29
4.8.2.1 Term Query 29
4.8.2.2 Prefix Query 29
4.8.2.3 Range Query 29
4.8.2.4 Function Score Query 29
4.8.3 Compound queries 29
4.8.3.1 Dis Max Query 29
4.8.3.2 Bool Query 30
4.8.4 Filter 30
4.9 Conclusion 30

5 Design **33**
5.1 Mapping 33
5.1.1 Languages 33
5.1.2 Data denormalization 33
5.2 Analyzer 34
5.2.1 English analyzer 34
5.2.2 Czech 35
5.2.3 Default analyzer 35
5.3 Query 35
5.3.1 Languages 35
5.3.2 Combining queries 36
5.3.3 Typos 36
5.3.4 Common words 36
5.3.5 Code 36
5.3.6 Name 36
5.3.7 Continent, Country and Category 37
5.3.8 Short description 37
5.3.9 Long description 37

5.3.10 Leader . . . 37
5.3.11 Location . . . 37
5.3.12 Organisation . . . 37
5.3.13 Results . . . 37
5.4 Indexing changes . . . 38
5.5 Security . . . 39
5.5.1 Network . . . 39
5.5.2 Search query . . . 39
5.6 Privacy . . . 39
5.7 Cluster . . . 39
5.8 Conclusion . . . 39

6 Realization **41**
6.1 Analyzers . . . 41
6.1.1 AnalyzerInterface . . . 41
6.1.2 Mapping generation . . . 42
6.2 Indexing . . . 42
6.2.1 Fields . . . 42
6.2.2 Project Document class . . . 42
6.2.3 Bulk indexing . . . 43
6.2.4 Project indexer class . . . 43
6.2.5 Index command . . . 44
6.3 Query construction . . . 44
6.3.1 Transformers . . . 44
6.3.2 Search service . . . 44
6.3.3 Search facade . . . 44
6.3.4 Pagination . . . 45
6.3.5 Tuning relevance . . . 45
6.4 Tools . . . 45
6.4.1 Git . . . 45
6.4.2 Kibana . . . 46
6.4.3 Xdebug . . . 46
6.4.4 Blackfire.io . . . 46
6.5 Conclusion . . . 47

7 Testing **49**
7.1 Manual testing . . . 49
7.2 Integration tests . . . 49
7.3 Static analysis . . . 51
7.3.1 PHP Parallel Lint . . . 51
7.3.2 PHPStan . . . 52
7.3.3 PHPMD . . . 52
7.4 Monitoring . . . 52
7.4.1 Prometheus . . . 52

7.4.2 Metrics . . . 53
7.4.3 Grafana . . . 53
7.5 Conclusion . . . 53

8 Possibilities 55
8.1 Analytics . . . 55
8.1.1 AB testing . . . 56
8.1.2 User signals . . . 56
8.2 UI . . . 56
8.2.1 Autocomplete . . . 56
8.2.2 Autoloading results . . . 56
8.3 Rerank . . . 56
8.4 Quality rank . . . 57
8.5 Preprocess query . . . 57
8.5.1 Taxonomies . . . 57
8.5.2 Translation . . . 57
8.6 Similarity tuning . . . 58
8.7 In-place update of donations . . . 58
8.8 Pagination without offset . . . 58
8.9 Cache . . . 58

Conclusion 59

Bibliography 61

A Glossary 69

B Contents of enclosed CD 71

List of Figures

2.1 Entity-Relationship Diagram (ERD) of HS project 9
2.2 Screenshot of current search 11

4.1 Custom analyzer - transformations [45] 26

5.1 Denormalized data of HS project 40

7.1 Graph of CPU usage during indexing 54

List of Listings

1 Example of a bool query 31
2 Example of an integration test 51

Introduction

HelpStone[1] (HS) is crowdfunding platform for non-profit organizations (NGOs). Its principle resembles Kickstarter[2], but instead of new products and services it helps fund projects which aim to make the world a better place. HS itself is a NGO and is funded by voluntary donations just like the projects it hosts.

"*HelpStone has made it its mission to help people in need all around the world, and to support education, the environment, endangered species, science, culture, sport, the arts, as well as every credible non-profit project.*" [1]

Since HS aims to accommodate many projects, it also needs to provide a good way for people to find a project which they are willing to donate their money to.

To do so, I am going to implement a new search module, which will help users find a project aimed at a cause they see as worthy of their money. I chose this as a topic for my bachelor's thesis because I am interested in search and I like the idea HS is built on.

Search is important because in contrast to browsing a site link by link, it is a lot more dynamic interaction which allows the user to find desired content a lot quicker. Unlike navigating a site link by link or by using Simple Query Language (SQL), user's fulltext queries are not an exact filter that simply matches a set of results. For example, when users search inside YouTube[3] for "cute cats" they are looking for videos that contain cute cats and aren't really interested in the video's title nor description. Video titled "silly kittens" might contain exactly what they want to see, while not containing any of the words they searched for.

[1] `https://helpstone.org`
[2] `https://kickstarter.com`
[3] `https://youtube.com`

In order to serve the users results they will be satisfied with, search connects multiple science disciplines together. For example, natural language processing to better match words that are related to each other, statistics to identify which items are better match for a query, distributed systems to be able to search on large data sets and many others.

When I use pronoun “we”, then I am talking about information or decision that comes from HS project and it is not subject of this thesis to verify or prove its correctness. When I use the pronoun “I”, then I am talking about what I have done while working on this thesis.

Upper indexes are references to footnotes, they contain links to entities mentioned in the text. Numeric indexes inside square brackets are citations and lead to bibliography.

Chapter 1

Goals

The ultimate goal is to provide users with the ability to find a project they will want to donate to. To achieve this goal I first need to summarize existing system in order to identify requirements. Based on this information I will compare different search engines, select best fit and implement new search based on it. This can be broken down into following tasks:

- Analyze requirements provided by HS.
- Summarize technologies HS uses to implement server side system and identify technological requirements.
- Summarize relational data model used for storing projects.
- Identify shortcomings of current search implementation.
- Compare existing search engines and find one fitting those requirements.
- Design data model and search queries for selected search engine.
- Implement new search module using selected search engine.
- Create a method to index current data to search engine and continuously index changes.

Chapter 2

HelpStone

In introduction I briefly explained HelpStone's mission. In this chapter I will describe HS more in depth, summarize it's technical aspects and identify requirements. Based on this information I will be able to select fitting search engine and design entire search module.

In this chapter I use certain words to signify requirement level. Those words are taken from RFC 2119 [2].

- MUST as "*an absolute requirement...*"
- MUST NOT as "*an absolute prohibition...*"
- SHOULD as "*there may exist valid reasons in particular circumstances to ignore a particular item...*"
- SHOULD NOT as "*there may exist valid reasons in particular circumstances when the particular behavior is acceptable or even useful...*"
- MAY as "*an optional item...*"

2.1 How HelpStone works

Knowing how HS works is necessary to understand what the users are likely to search for and what the data they search in look like.

NGOs can create multiple projects for different activities they do. Every project has its own page with detailed information and a credit card gateway which people can use to send their donations. All project pages also contain contacts, which lead directly to the organisation, so anyone can get their questions answered directly from the organisation. NGOs can also publish their bank account information on their project page, which allows people to bypass HS

altogether and send their donations directly to their chosen organisation. However international bank transfer fees are costly and sending money via HS using a credit card saves more money for the actual project.

When users send a donation of certain amount via HS, they receive an actual HelpStone stone on a bracelet, with unique code of the project they donated to, as a reminder of their good deed. More information about HelpStone can be found at `https://helpstone.org`.

An ideal example, of a non-profit organisation HelpStone supports, is `greenbooks.org`. Their mission is promoting eco-literacy by creating libraries and children's books.

2.2 Software

In this section I will summarize software, programming languages and frameworks used by HS and formulate technological requirements based on my findings.

2.2.1 Docker

Docker[4] is software for running applications and services inside containers, an alternative to visualizing entire machine. All software is run inside multiple connected docker containers. Docker runs linux and itself runs on linux.

To integrate well into existing technological stack, new search solution *MUST* be linux and docker compatible.

2.2.2 MySQL

All data is stored in relational database MySQL[5] version 5.7 using InnoDB[6] engine.

Search module *MUST* be able to work with data stored inside MySQL.

[4] `https://www.docker.com/`
[5] `https://www.mysql.com/`
[6] `https://dev.mysql.com/doc/refman/5.7/en/innodb-storage-engine.html`

2.2.3 PHP

Language used for server side programming is PHP[7] version 7.1. Newer versions of PHP, specifically 7.0[8] and 7.1[9], introduced important features into PHP.

Search module *MUST* be built using PHP, and *SHOULD* use new features from version 7.0 and 7.1 in order to achieve higher code quality.

2.2.4 Symfony

Application is written using Symfony framework[10] version 3.4. Which is an OSS framework for creating web applications.

Search module *MUST* use Symfony tools, specifically its dependency injection container.

2.2.5 Doctrine 2 ORM

Doctrine ORM is Object Relational Mapper[11] (ORM) used to map PHP objects to relational database's tables. It is an abstraction over Doctrine DBAL[12], which is a language abstraction over SQL.

In order to keep database abstracted just like the rest of the application, search module *SHOULD* use Doctrine Query Language[13] (DQL) to gather data from the database.

2.2.6 RabbitMQ

RabbitMQ[14] is a message queue implementing the Advanced Message Queuing Protocol[15] (AMQP). HelpStone utilizes RabbitMQ using publisher-subscriber pattern. When user performs certain actions, new message is published and subscribers waiting for different messages then asynchronously perform tasks like sending emails, generating image thumbnails and processing donations.

Search module *MAY* use it if a need arises.

[7] `https://php.net`
[8] `http://php.net/releases/7_0_0.php`
[9] `http://php.net/releases/7_1_0.php`
[10] `https://symfony.com`
[11] `https://www.doctrine-project.org/projects/orm.html`
[12] `https://www.doctrine-project.org/projects/dbal.html`
[13] `https://www.doctrine-project.org/projects/doctrine-orm/en/2.6/reference/dql-doctrine-query-language.html`
[14] `https://www.rabbitmq.com/`
[15] `https://www.amqp.org/`

2.2.7 Redis

HS is using Redis[16] as a cache, to store data expensive to compute.

"*Redis is an open source (BSD licensed), in-memory data structure store, used as a database, cache and message broker. It supports data structures such as strings, hashes, lists, sets, sorted sets with range queries, bitmaps, hyperloglogs and geospatial indexes with radius queries. Redis has built-in replication, Lua scripting, LRU eviction, transactions and different levels of on-disk persistence, and provides high availability via Redis Sentinel and automatic partitioning with Redis Cluster.*" [3]

Search module *MAY* use it if need arises.

2.3 Data model

In this section I describe data and its relations. Overview can bee seen in a diagram in fig. 2.1.

2.3.1 Translations

As mentioned earlier, HS is an international project. To allow people from all over the world to use HS, it is desirable to translate projects into a language they understand. Each project can be translated into any language, while english translation is always required. This means that any information described in this section can always exist in multiple language versions. Excluding things that are not usually translated like names, addresses and numbers. In further text "project translation" or "translation" refers to project representation in specific language, while "project" refers to entire project with all its translations and metadata.

2.3.2 Metadata

Projects contain metadata like continent, country, location name with address and category. Although on most of these users can search using a filter, search *SHOULD* also try to match them using full-text search. For example, query "czech education" should also match a project with country "Czech Republic" and category "education".

[16] `https://redis.io/`

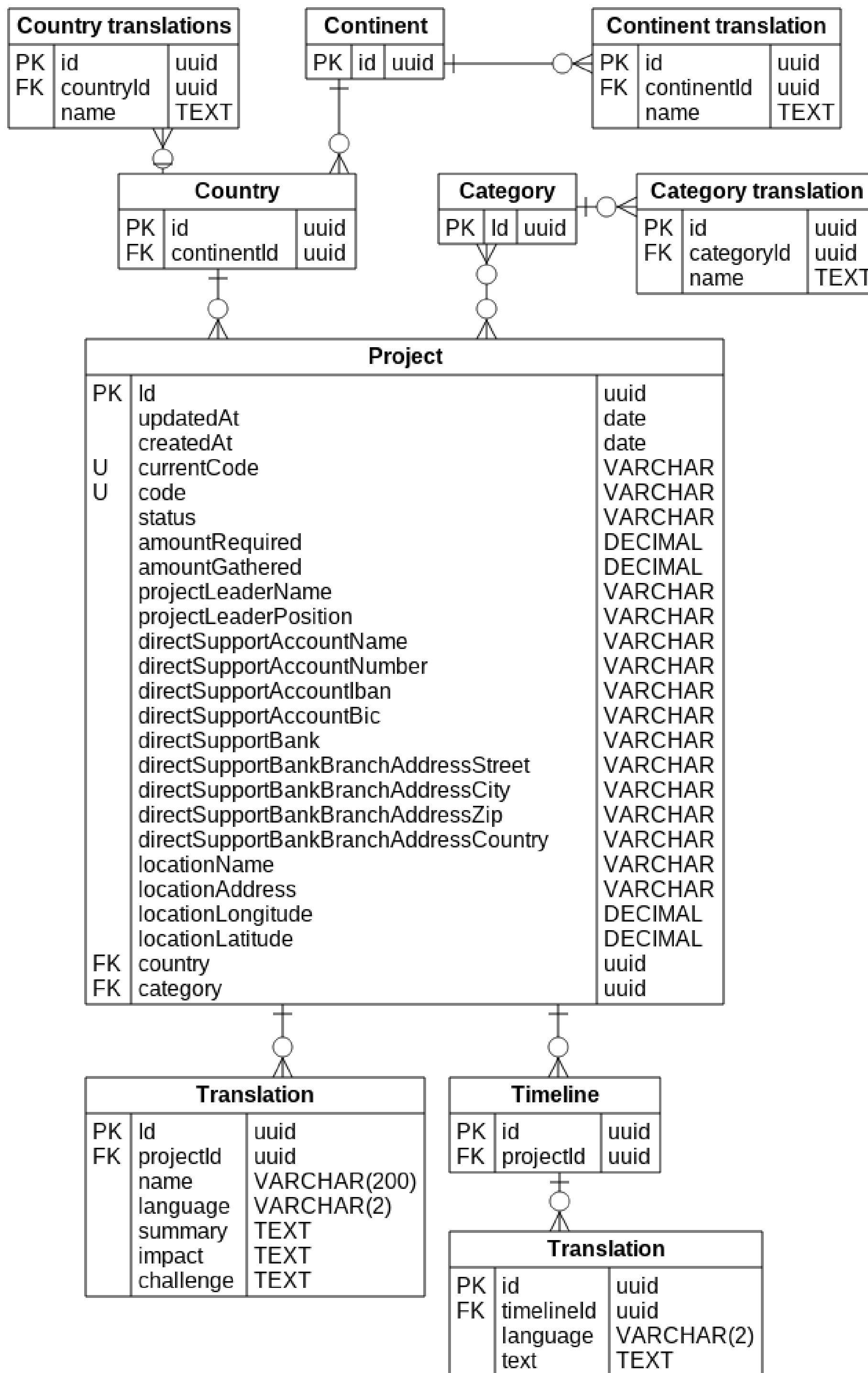


Figure 2.1: Entity-Relationship Diagram (ERD) of HS project

2.3.3 Text fields

In this subsection I describe text properties of a project. All of which *MUST* be searchable using full-text search.

Information about organisation behind the project.

- Name
- Website
- Phone
- Contact person

Information about the project.

- Name
- Short description (displayed in lists)
- Long description
- Direct bank account information (if made public by the NGO)
- Project leader - name and title of person leading the project

2.3.4 Timeline

Timeline entries are short messages comparable to Twitter[17] Tweets or Facebook[18] status updates. They can also be translated into other languages. HS is trying to motivate NGOs to share updates on their project via timeline posts to keep donors updated on progress their donations helped achieve.

Timeline entries *MUST* be searchable using full-text search.

2.3.5 Size

HelpStone is multilingual and in theory each project can be translated to any and all of existing languages. At the moment HS project recognizes 184 languages.

Based on estimation made by HS project, new search module *MUST* be able to handle 4000 projects, each having 5 translations on average. Typical project translation is expected to have about 4000 characters.

[17] `https://twitter.com`
[18] `http://facebook.com`

Figure 2.2: Screenshot of current search

2.4 Current search

Current search allows users to input their query into single text field and check several checkboxes to further narrow their query results. In fig. 2.2 is screenshot of current search user interface.

2.4.1 Features

Current search implementation can:

- Find direct, letter to letter, text matches of entire user query.
- Filter based on category, continent, country and raised money.

2.4.2 Missing features

Current search implementation cannot:

- Sort by relevance, all returned results have the same relevance to user's query.
- Find text that is relevant to the query but is not direct, letter to letter, match to user's query. For example, query "animals" does not match text "animal".

2.4.3 Implementation

Current implementation uses SQL queries to find results. It first concatenates all searchable columns into one and then searches it using the "like" operator[19]. Because it uses concatenation and like clause each search, query scans entire text which cannot be sped up using index.

2.5 Requirements

In this section I summarize search requirements imposed by HS project.

2.5.1 Functional requirements

In context of this thesis, functional requirements primarily define what data should the user be able to search in and how should the search behave.

2.5.1.1 User interface

Current user interface UI was designed to fit the website and not be cluttered with too many options. New implementation *MUST NOT* change existing search UI.

2.5.1.2 Stone

Highest quality projects get assigned a stone. User *MUST* be able to search projects that do or don't have it.

[19] https://dev.mysql.com/doc/refman/5.7/en/string-comparison-functions.html

2.5.1.3 Code

Code is Unique identifier (UID) of a project hosted on HS. There are two types of codes, each project has one five-letter code and if a project has been assigned a stone, it also has a four-letter code from the stone (described in section 2.1). This code is physically on HS stones and people need to be able to find a project based on it. When user's query matches a code, it *MUST* be the first result, or if it is only result user *MUST* be redirected directly to the project page.

2.5.1.4 Date

Projects are marked with date they were created and date of the last update. Search module *SHOULD* allow searching for projects created or updated in a certain time period.

2.5.1.5 GPS coordinates

Although current search UI does not support searching projects based on GPS location or a "search near me" function, it is a feature that is considered for the future. Search module *SHOULD* be able to work with GPS coordinates.

2.5.1.6 Filtering

Filters generally work as yes/no query, removing all items that do not match. For example, with filter for categories "education" and "children" only projects matching both categories will be returned.

These filter queries often filter out results unfairly because it excludes items immediately behind the edge of filter, which the user might still find relevant. In this case results matching both categories SHOULD be before results matching only one category. It is only *SHOULD* because filtering is usually used with a search text, in which case items matching text and one category *SHOULD* be returned before a project matching only categories. Items matching no category *SHOULD NOT* be returned.

2.5.1.7 Languages

Since HS is multilingual, the search module MUST support searching in all different translations of a project. To store multilingual data search module MUST be able to work with UTF-8 text.

Because users likely don't speak all languages, search *MUST* prefer projects which have a translation in a language the user understands.

2.5.1.8 Synonyms

Some words have synonyms, search SHOULD be able to match projects using synonyms. For example, “Czech Republic” and “Czechia” are according to the United Nations Terminology Database [4] long and short versions of our country’s name. Other possible synonyms are country ISO codes “CZE” and “CZ” [4]. Therefore searching for one SHOULD have same results as searching for the others.

Another case are words that do not represent same thing, but are related or people often confuse them. An example would be “tornado” and “hurricane” which differ in size, speed, and several other attributes [5] but are similar in damage and appearance. Search *SHOULD* be able to handle synonyms.

2.5.1.9 Typos

Users often mistype a word or make a spelling mistake. Search *SHOULD* be able to handle simple typos.

2.5.1.10 Diacritics

Some languages, like czech, make use of diacritics. However it is common for users, especially on mobile devices, to type words without diacritics. Search *SHOULD* be able to match words with diacritics to word without it. For example, word “český” should match word “cesky”.

2.5.2 Non-functional requirements

2.5.2.1 Sensitive data

NGO’s privacy must be respected. Search module *MUST* index and search only in data that are publicly accessible via HS website. Any other data *MUST NOT* leave the HS database.

2.5.2.2 Technology compability

Search module *MUST* use only technologies that are compatible with current HS technology stack, specified in section 2.2. HS prefers open source technologies.

2.5.3 Data size

Search module must be able to handle data of size described in section 2.3.5.

2.5.3.1 Maintainability

Technology used to build the search module MUST be stable.

- *MUST* have a table release.
- *MUST* be used by established projects.
- *SHOULD* be backed by a company that actively develops it.

2.6 Conclusion

I went over current state of search, what is working and isn't working, explored used technologies and identified requirements. Now, based on this information, I must find a search engine which will allow me to create a search module fulfilling these requirements.

Chapter **3**

Existing technologies

In this chapter I go over several search engines, describe their main features and differences to other search engines. And in conclusion (section 3.5) I select a search engine which I will use to implement new search module.

3.1 MySQL

MySQL has support for full-text search using the *fulltext* index and the *match* function[20] while querying.

MySQL splits text into words and indexes them using inverted index. However it does not perform any language analysis, such as removing affixes or diacritics [6]. Its search language is fairly simple and I deem it unsatisfactory for my use case [7].

It is however possible to perform text analysis like stemming or diacritics removal on application level and then index and query preprocessed text.

3.2 Lucene Core

Apache Lucene Core is "*a high-performance, full-featured text search engine library written entirely in Java. It is a technology suitable for nearly any application that requires full-text search, especially cross-platform.*" [8]. It is licensed under the Apache License, Version 2.0 [8].

[20] https://dev.mysql.com/doc/refman/8.0/en/fulltext-search.html#function_match

According to [8], some of its attributes are:

- "*index size roughly 20–30% the size of text indexed*"
- "*many powerful query types: phrase queries, wildcard queries, proximity queries, range queries and more*"
- "*fielded searching (e.g. title, author, contents)*"
- "*allows simultaneous update and searching*"

Lucene itself is only a search library [8] which other tools use to perform search. It is possible to use Lucene directly inside an application, but it requires the developer to solve a lot of problems which can be solved using an existing search engine, which offers a higher level API. Search engines based on Lucene are, for example, Elasticsearch described in section 3.3 and Solr described in section 3.4.

3.3 Elasticsearch

Elasticsearch[21] is distributed [9] search engine server based on Apache Lucene [10].

"*Elasticsearch is a highly scalable open-source full-text search and analytics engine. It allows storing, searching, and analyzing big volumes of data quickly and in near real time. It is generally used as the underlying engine/technology that powers applications that have complex search features and requirements.*" [11].

It is licensed under the Apache License 2.0 [12]. While being OSS, it is also backed by a company named Elastic[22]. Elastic offers additional plugins and licenses with support [13]. Elastic also maintains documentation[23].

Elasticsearch provides language analysis out of the box with prepared language analyzers [14]. It also provides complex, JSON based, query language [15].

Elasticsearch is linux compatible and can run inside Docker using an official image [16]. PHP libraries for Elasticsearch exist, for example Elastica[24]. Elasticsearch is compatible with non-functional requirements defined in section 2.5.2.

[21] `https://www.elastic.co/products/elasticsearch`
[22] `https://elastic.co`
[23] `https://www.elastic.co/guide/index.html`
[24] `https://github.com/ruflin/Elastica`

3.3.1 Maturity

To asses how mature Elasticsearch is, I looked for companies that use it in production for years.

ThoughtWorks[25] marked Elasticsearch as production ready in their Technology Radar in 2013 [17].

GitHub[26] is using Elasticsearch to index over 8 million code repositories with over 2 billion documents, serving 300 search requests per minute on average [18].

In 2014 Wikimedia moved to Elasticsearch and uses it for all their sites. As one of the reasons why they prefer Elasticsearch to other tools they write "*Elasticsearch's super expressive search API lets us search any way we need to search and gives us confidence that we can expand on it. Not to mention we can easily write very expressive ad-hoc queries when we need to.*" [19].

"*The New York Times put all 15 million of its articles published over the last 160 years into Elasticsearch*" [20].

3.4 Solr

Apache Solr[27] is a search engine server and is a part of the Lucene project and uses Lucene Core to perform search.

"*Solr is highly reliable, scalable and fault tolerant, providing distributed indexing, replication and load-balanced querying, automated failover and recovery, centralized configuration and more. Solr powers the search and navigation features of many of the world's largest internet sites.*" [21]

Solr is an Apache Foundation [21], which is a strong argument for project stability.

Solr offers out of the box language analyzers [22]. It also offers several query languages *Standard*, *DisMax* and, *Extended DisMax* [23, 24, 25].

Solr is linux compatible and can run inside Docker[26]. PHP libraries for Solr exist, for example Solarium[28]. Solr is compatible with non-functional requirements defined in section 2.5.2.

[25] `https://www.thoughtworks.com`
[26] `https://github.com`
[27] `http://lucene.apache.org/solr/`
[28] `https://github.com/solariumphp/solarium`

3.4.1 Maturity

To asses how mature Elasticsearch is, I looked for companies that use it in production for years. Solr's list does not contain information how long each company uses it. Solr is, for example, used by Nasa, Netflix, Disney and Apple [27].

3.5 Conclusion

Based on descriptions and users of these search engines, I chose Elasticsearch. It fulfils all non-functional requirements specified in section 2.5.2 and has all the necessary features to implement functional requirements specified in section 2.5.1.

To be able to design and implement search module using Elasticsearch, I need to explore it more in depth, which I will do in the next chapter.

Chapter 4

Elasticsearch theory

I briefly described Elasticsearch in chapter 3, in this chapter I go more in depth and into concepts behind it.

4.1 Inverted index

Usual representation of text document is an ordered list of words. To find out in which documents a word is present, one has to traverse entire text of each document and try to find the word. An inverted index takes different approach. Each word has a list of document IDs in which the word occurs [28]. Searching for documents that contain a set of words means traversing the index for each word and finding document IDs common for all of the words.

4.2 Relevance

Relevance is used to measure quality of search. A "relevant result" or "relevant item" to a query is an item that the users would like to see when they search. An "irrelevant result" is an item that is returned, but the users think it is irrelevant.

To measure how good a search is, there are several metrics. The most common being recall and precision. Following measurements expect we have manually marked what items are relevant for our query.

4.2.1 Recall

Recall indicates how many relevant items are missing from results. A search engine that would return all items in the database for each search would have 100% recall, because all relevant items would be present for sure.

4.2.2 Precision

Precision indicates how many items from results are relevant. A search engine, that would return only single and relevant item, would have precision 100%.

4.2.3 Recall and Precision relation

Good search tries to maximize both these metrics. Having returned all relevant results and only relevant results returned is the perfect result.

4.3 Ranking and Scoring

To allow high recall, while not compromising precision, searches usually implement result ordering, also called ranking. Rank of an item inside results is usually based on a score. This puts more relevant items before the less relevant. Score is numeric representation of how much an item is relevant to the query. For example, when searching "endangered animals in Africa", document which contains all of these words will have higher score than a document that contains only words "animals" and "Africa". Score is computed based on term frequencies.

4.4 Term frequency

Simple inverted index described in previous section can only answer whether there is a word present inside a document, yes or no question. But in text, different words have different importance. To measure importance of words, search typically uses statistics of word frequencies.

The statistics are usually computed for two different cases, for each field in document and for field across entire index [29]. The more times a word is present in a field in document, the more it is important to the document. The more documents in index contain the same word in the field, the less it is important overall.

For example, words "the", "and", "or" are common for most english texts, therefore it is not that important for each text. The word "tiger" is uncommon, therefore tiger is important to the document.

Elasticsearch uses BM25, which is an algorithm that computes a score based on term frequencies [30].

4.5 Cluster

Elasticsearch is a distributed system, which allows it to scale horizontally the number of searches and the amount of data searched.

Running Elasticsearch instances are called nodes, they are connected over a network and together they form a cluster. Adding more nodes means adding new servers.

To scale an index in Elasticsearch cluster, data are stored in shards [31]. Shards are distributed inside cluster in order to utilize hardware. A shard is lucene index and Elasticsearch uses two types of shards, primary and replicas.

Primary shards split all the incoming data between themselves and searching then searches in all of them and combines the results. Having more primary shards allows for storing and searching large data.

Replica shards are copies of primary shards [32]. Adding more replica shards allows for cluster to work after a primary shard fails. Also replicas can handle search requests, so adding more replicas increases the number of searches cluster can handle.

Because the amount of data is not expected to outgrow a single primary shard in near future, clustering is not further looked into in this thesis. More information about Elasticsearch scaling can be found at `https://www.elastic.co/guide/en/elasticsearch/guide/current/scale.html`.

4.6 Data types

Elasticsearch stores its data as documents, using JSON as a representation [33]. Because JSON type system is simpler than the Elasticsearch type system, there is also a separate way to specify types for fields. If a mapping is not specified, Elasticsearch guesses the type and creates mapping automatically.

Once a mapping for a field is set, it is impossible to change it. It is possible only to add a new field. If changing a field is required then reindexing is necessary.

In following sections I describe several important data types.

4.6.1 Text

This data type is used to store analyzed text for full-text search [34].

4.6.2 Keyword

Keyword type is not analyzed and is not full-text searchable [35]. It is intended for matching entire field content using term queries[29].

4.6.3 Numeric

Numeric data type is used to store numbers, similar to programming languages it offers types of different length like "long", a signed 64-bit integer or "double", a single-precision 32-bit IEEE 754 floating point number [36].

4.6.4 Boolean

Boolean is used to store true/false values [37].

4.6.5 Date

Used to store date and time with millisecond precision [38].

4.6.6 Geo-point

Storing geographical coordinates is done using geo-point, which is data type composed of longitude and latitude [39].

4.6.7 Object

Elasticsearch is a document database and because document database doesn't follow normalization forms known in relational databases, it keeps all data inside the document. In case of Elasticsearch documents are represented by JSON and it can contain inner objects. This data type indicates that a field itself is an object [40].

4.6.8 Array

In Elasticsearch every field can be used as an array, so it does not have a dedicated array data type [41].

4.6.9 Nested

"*The nested type is a specialised version of the object data type that allows arrays of objects to be indexed and queried independently of each other.*" [42]

[29] `https://www.elastic.co/guide/en/elasticsearch/reference/6.2/term-level-queries.html`

4.7 Analysis

Analysis is a process of converting text into more searchable form [43]. The most basic step is splitting text into words. Additional transformations are usually applied like removing HTML tags, removing diacritics, converting words to their base form. Analysis is performed both at indexing and searching, the analyzer is usually same both times [44]. Analysis is performed in steps that take output from one step as an input to another An example analyzer can be seen in fig. 4.1.

4.7.1 Character filters

"*A character filter receives the original text as a stream of characters and can transform the stream by adding, removing, or changing characters.*" [46]

4.7.1.1 HTML filter

HTML filter is a character filter, which removes non-visible HTML content such as tags with attributes, leaving only their text content [47]. For example, "<p class="cls">Some text</p>" will be transformed to "Some text".

4.7.2 Tokenizer

Tokenizer's role is to split text into tokens [48]. Splitting is usually meant to split text by words, using whitespace or word delimiting characters like comma or a dot. Splitting is also possible based on a specific pattern, for example IP address.

4.7.3 Token filters

Token filters process a stream of tokens and try to transform each token in a way that makes it more suitable for searching [49].

4.7.3.1 Lowercase filter

Lowercase filter is a token filter that normalizes token to lower case [50]. A token "Prague" will be transformed to "prague"

4.7.3.2 ASCII folding filter

ASCII folding filter tries to transform non-ASCII characters to similar ascii character if one exists [51]. For example, "À", "Á", "Â", "Ã", "Ä", "Å" all would get transformed to "A". It also have a parameter "preserve_original" which duplicates the token and transforms only one of them, leaving the original intact.

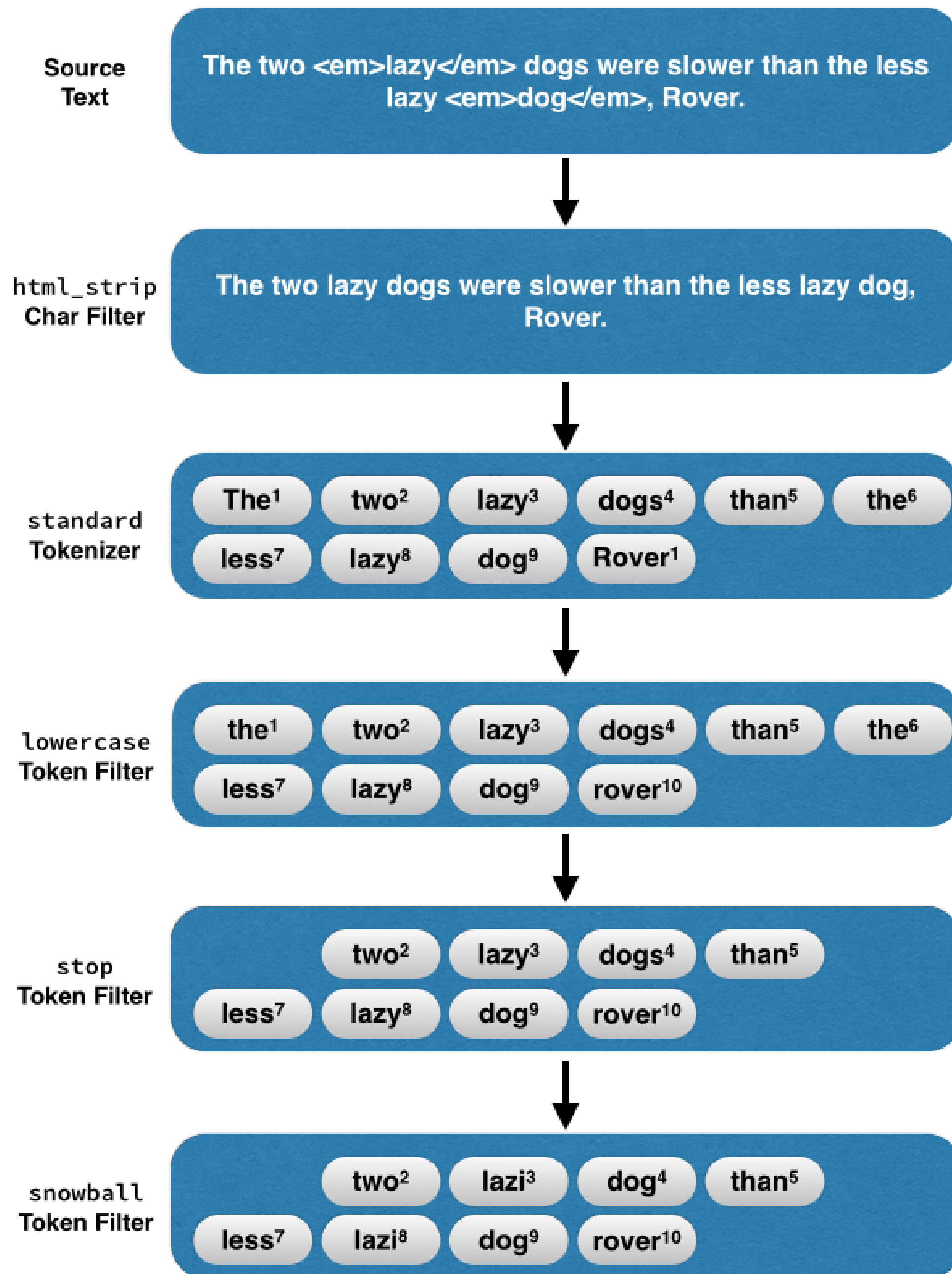


Figure 4.1: Custom analyzer - transformations [45]

4.7.3.3 Stemmer filter

Stemmer token filter tries to reduce a word to its base form [52]. For example by removing prefix and suffix. Stemming usually implies algorithm based on a language specific set of rules. These rules are not perfect and sometimes they stem two different words into one that means something different.

Words “argue”, “argued”, “argues”, “arguing”, and “argus” will get transformed into word “argu”. While the first four words are related, argus is a bird or giant in Greek mythology.

It also has trouble matching irregular words such as “goose” and its plural “geese”. Those get stemmed as “goos” and “gees”, which means searching for one will not match the other. There is also a dictionary based stemming, usually called lemmatization, which has its own downside discussed in section 4.7.3.4.

4.7.3.4 Hunspell filter

Hunspell token filter provides dictionary stemming, usually referred to as lemming [53]. Because it is dictionary based, this filter is only as good as its dictionary. Uncommon, new or slang words will stay unanalyzed.

4.7.3.5 Keyword repeat filter

This filter takes a token stream and outputs each token twice, once as a keyword and once as a non-keyword [54]. A keyword marked word will not be stemmed and will be indexed alongside the unstemmed word.

4.7.3.6 Unique filter

Unique filter is a token filter that removes duplicates from analyzed text. It has an option only_on_same_position which removes duplicates only if they are on the same position in text [55]. This is useful when different filters create duplicates, for example, when using a keyword repeat filter with a stemmer stemmer that doesn’t know how to stem the word.

4.7.3.7 Synonym filter

Synonym filter is a token filter that handles searching for synonyms [56]. There are two ways it can work, one is by indexing all the synonyms at the same position in text, the other is converting all the synonyms to the same word.

4.7.3.8 Stop filter

Stop token filter removes stop words based on a list of stop words [57]. Stop words are words that occur so often they loose meaning for search. The top stop words in english language are the articles “a”, “an” and “the” and short and common words “to”, “be”, “not”. They occur in all non-trivial texts and searching for a word “the” will match virtually everything. That is why it is common for search engines to ignore these words, so searching for “the prize” is the same as searching for “prize”.

Stop words have downsides, some phrases are made entirely of stop words and will match nothing, even though they have a meaning. An example of this is “to be or not to be” which will get filtered to an empty string. Other cases are phrases or names of organisations, where these words are important. In different contexts different words are too common. For example, searching “video about XYZ” is as useful as searching for “XYZ”.

4.8 Queries

In this section I introduce different query types, which are used to construct Elasticsearch query.

4.8.1 Text queries

These queries are designed for full-text search. They utilize analyzers and compute a score.

4.8.1.1 Match Query

According to [58] match query uses analyzers on text it is given and constructs term queries from the analysed text. It can be specified how many words must match minimum, which allows to tune precision. It can handle typos using *fuzziness*, which generates words within specified edit distance. It also offers *cutoff frequency*, which is an alternative to stop words. It dynamically removes common words (those with high frequency), which is an advantage over a stop word filter, which requires predefined list of words, which are language and domain dependent. It can be set what frequency should be considered too high.

4.8.1.2 Phrase Query

Similar to match query it first analyzes text, but when searching it looks only for documents containing all of the searched words and in the same order as in the query [59, 60].

4.8.2 Term queries

Term queries operate on the entire field, using it as one value.

4.8.2.1 Term Query

Term query looks for a field with exact match [61]. It is analogous to *where field='value'* clause in SQL.

4.8.2.2 Prefix Query

Prefix query is similar to term query, but instead of exact match it is looking for matching prefix [62]. That is if the field contains the searched string as a prefix.

4.8.2.3 Range Query

Range query is used for searching on fields that can be ordered, like numbers or dates [63]. It can search with operators less-than "lt", less-than or equal "lte", greater-than "gt" and greater-than or equal "gte". It can operate on numbers, dates and lexicographically on strings [64].

4.8.2.4 Function Score Query

Function score query allows modifying the score of returned documents [65].

For example, using script score I can perform mathematical operations on numeric fields and boost accordingly.

Another one is decay function which I can use to make documents further from a coordinate or a date less important.

4.8.3 Compound queries

Compound queries are used to combine queries together [66]. They can combine other compound queries as well.

4.8.3.1 Dis Max Query

"*A query that generates the union of documents produced by its subqueries, and that scores each document with the maximum score for that document as produced by any sub-query*" [67]

This is useful when searching the same text multiple times, using different analyzers or queries (usually with different boosts) to find only the best match.

4.8.3.2 Bool Query

Bool query combines other queries using four different approaches “must”, “must not”, “should” and “filter” [68]. Queries from “should” do not have to match, they only add score. Queries from “filter” do not compute score, they are true/false filter. All queries in “must” must match. All queries from “must not” must not match. Scores from “must” and “should” are added together.

The following query finds all documents where “firstName” is “John”, lastName is “Smith” with age between 18 and 30. Additional score will be added to people which are alive. This query in bool representation in json format can be seen in listing 1.

4.8.4 Filter

Elasticsearch recognizes two search contexts, query and filter [69].

Query context computes scores and sorts results based on it, returning only results with non-zero score. Searching for “pink panther” will match items that contain both words or one of the words, order of the words is not taken into account. Advantage of query context is its high recall and ability to sort results. The main disadvantage is speed. It also has lower relevance, but since the results are sorted, all the non-relevant items are in the back.

Filter context does not compute score, it is evaluated as yes/no query, removing items from results that do not match. Searching for “pink panther” will match fields that are exactly “pink panther”. Advantage is its high relevance because only items containing direct matches are returned. Since it uses direct matches and no score, it is fast to compute and Elasticsearch is able to use cache. Disadvantage is it’s low recall, where text “popular panthers are pink and black” will not be matched.

4.9 Conclusion

Now that I explored how Elasticsearch works, including how it saves data and the language used to query it, I can use this knowledge to design search specifically for HS, which I will do in the next chapter.

```
{
  "query": {
    "bool": {
      "must": [
        {
          "term": {
            "firstName": "John"
          }
        },
        {
         "term": {
            "lastName": "Smith"
          }
        }
      ],
      "filter": {
        "range": {
          "age": {
            "gte": 18,
            "lte": 30
          }
        }
      },
      "should": [
        {
          "term": {
            "deceased": 0
          }
        }
      ]
    }
  }
}
```

Listing 1: Example of a bool query

Chapter 5

Design

In this chapter I take theory about Elasticsearch from chapter 4, combine it with knowledge about HS from chapter 2 and design a new search module.

5.1 Mapping

In section 4.6 I described several field types Elasticsearch supports. In this section I am going to design a mapping to store HS projects.

5.1.1 Languages

Most important decision is how to handle languages, that is how to save project translations. There are two prevalent ways, having separate index for each language, issuing same query to all of them when searching, or saving it inside single index and having a special query that is aware of languages.

After researching pros and cons of each way and a discussion with my supervisor I decided to use single index, because we expect it will allow me to spend more time working on query quality instead of configuring indexes.

Converting the single index solution into the multiple index one is fairly straight forward. Create an index for each language and in each index only create fields that are specific to the language or are language agnostic. Elasticsearch handles non-existing fields gracefully, simply anything trying to match it returns no match. My query is designed to work even in this case.

5.1.2 Data denormalization

As I wrote in section 2.2, all projects are stored inside MySQL, which is relational database. Project data is stored in normalized form in multiple tables, connected using foreign keys. To store this data inside a document

store like Elasticsearch I first must denormalize the data into single document (JSON). The data was described in section 2.3 and can be best viewed in fig. 2.1.

I denormalized data by adding prefixes and storing them all in a single document. Denormalization algorithm is as follows:

- Take field name
- Prefix it with table name
- If it is “to many” relation with different languages, duplicate it and prefix each duplicate it with a language code
- If it is “to many” relation without language, map it as an array

Denormalized mapping with a translation to English and Czech can be seen in fig. 5.1.

5.2 Analyzer

In this section I describe analyzers for two currently most common languages used in HS. Using those analyzers fulfills functional requirements, described in section 2.5.1, for handling synonyms and diacritics. Using tokenizer fixes a flaw of current implementation, finding only direct matches, described in section 2.4.2.

5.2.1 English analyzer

- HTML character filter
- Standard tokenizer
- Lowercase filter
- Czech synonyms filter
- Keyword repeat filter (to keep unstemmed version)
- Czech stemmer filter
- ASCII filter
- Unique filter (only on same position, to remove unstemmed tokens)

5.2.2 Czech

- HTML character filter
- Standard tokenizer
- Lowercase filter
- English synonyms filter
- Keyword repeat filter (to keep unstemmed version)
- English stemmer filter
- ASCII filter
- Unique filter (only on same position, to remove unstemmed tokens)

5.2.3 Default analyzer

To provide search for other languages, I use a default analyzer, that should provide basic analysis for all languages that use whitespace to delimit words.

- HTML character filter
- Standard tokenizer
- Lowercase filter
- English synonyms filter
- ASCII filter

5.3 Query

I described several different query constructs in section 4.8. In this section I design a search query based on them.

5.3.1 Languages

Each project can have a translation into many, possibly all, languages. Which means a translation is different representation of the same project. Therefore it is not desirable to combine score from different translation, but to use only the score from best matching translation. This prevents me from using multi match query, which could result in combining score from different languages (English title, Czech short description, German long description). To solve this, the root query contains Dis Max Query (section 4.8.3.1) which has subqueries for each language and keeps only score from best matching language.

I used information from HTTP header, about what languages the user prefers, to boost Dis-Max subqueries for those languages.

Because Elasticsearch performs analysis for each full-text field searched, searching always all languages will be CPU intensive task. Should it become a problem, it is possible to search only in fields in languages the user understands, based on HTTP headers.

5.3.2 Combining queries

Because this is completely new implementation using different technologies, I decided to optimize first for recall, second for ranking and last for precision. This means I do not use yes/no filters, but instead combine score from all fields, showing as results all items that match at least one thing. This is achieved using two nested bool queries (described in section 4.8.3.2). Putting all text queries into *should* clause of one bool query, which combines score from all of them, but doesn't require them all to match. This bool query is inside *must* clause of the other bool query, which requires it to match at least one of its text queries. Otherwise all items would be returned even if they don't match any text.

5.3.3 Typos

I am going to handle typos by using automatic fuzziness in all match queries.

5.3.4 Common words

To remove common words, instead of using stop words filter, I am going to set cutoff frequency for match queries, so that common words are identified automatically.

5.3.5 Code

To perform exact match I shall use term query. Code is very short, which means there could be many false-positives when using non-exact match. Code is also very important, because it is unique identifier, which means it should have the highest boost.

5.3.6 Name

It should be analyzed and searched using match and match phrase. Project name is short and important, it should have the second highest boost.

5.3.7 Continent, Country and Category

I will implement search using match, to allow query like “czech children” match country “Czech Republic” and category “children”. Because these fields have limited number of values (are enumerable) and are short, they should have third highest boost.

5.3.8 Short description

Because it is a free-form text, it will be matched using match query. Short description is a text displayed in project lists, for example in search results. Because it is the second text after project name the user sees, it must have higher boost than the long description.

5.3.9 Long description

Because it is a free-form text, it will be matched using match query. Long description is a text displayed only on project detail page. Because it is the longest and free-form text, it should not be boosted.

5.3.10 Leader

Leader has name and position, both will be searched using match query. Position is less important than name, so it has lower boost.

5.3.11 Location

Location has name and address, both will be searched using match query.

5.3.12 Organisation

Organisation name, address and website will be searched using match query. Phone number will searched using prefix query.

5.3.13 Results

Elasticsearch allows retrieving entire document and showing it to the user, bypassing MySQL entirely. Unfortunately Elasticsearch has eventual consistency, but HS needs the user to always see up-to-date project, mainly because of amount of donated money, which needs to be always current. To achieve that I only load project IDs from Elasticsearch and then load current data from MySQL. This also allows me to completely reuse current template for search results.

5.4 Indexing changes

In this section I describe how new projects or changes to existing ones are saved to Elasticsearch.

When published project is updated or created, corresponding event is dispatched using the Symfony Event Dispatcher[30]. Listener takes this event and publishes new message with request to (re)index a project into RabbitMQ queue.

Consumer waiting for messages in queue takes the message, loads project from database and sends update to Elasticsearch. To speed things up, if there are multiple messages in the queue, consumer takes more of them and sends them to Elasticsearch as a bulk request, described in section 6.2.3.

Using event listeners decouples search from project creation/update. Service creating/updating projects only dispatches event (notification) that something happened.

The event is dispatched after database commit, which prevents consumer from trying to index something that was not yet saved in the database. However this creates eventual consistency, when Elasticsearch has saved an older version of a project, user can still search for it. This is not big problem, because Elasticsearch is used only to find IDs of projects and users are always displayed current version from DB, so in worst case user is displayed an irrelevant, but up-to-date, project.

This window of inconsistency can be kept very small by scaling number of consumers. Advantage of using queue is asynchronicity, if there was no queue, then indexing the project would slow down the HTTP request. In worst scenario, when search server is unavailable it would not index the changes, when using queue, messages will get processed once the search server is available again.

This eventual consistency also allows for multiple consumers to index changes for the same project at the same time. This can occur when a project is changed multiple times in short interval.

To make sure the newest change is indexed last (isn't overridden by an older version), Elasticsearch provides optimistic locking [70]. It allows documents

[30] `https://symfony.com/doc/3.4/components/event_dispatcher.html`

to be marked with a version number and when updating a document, Elasticsearch checks that the changes contain version number higher(newer) than the one present in the indexed document.

5.5 Security

5.5.1 Network

Elasticsearch servers accessible directly via network are vulnerable to malicious attacks, because Elasticsearch does not have user authorization built in. It is possible for attackers to download data, delete it from Elasticsearch and request a ransom. The recommended way to secure Elasticsearch against these attacks is to use private IP, accessible only from whitelist of clients [71].

Those whitelisted servers must construct queries themselves and not allow the users writing their own Elasticsearch queries and only proxying them.

5.5.2 Search query

Sometimes there is user input that is malicious and modifies the query to do something it should not. To make sure users aren't able to alter queries in a way they are not supposed to, all requests, (body, headers, url) are constructed from predefined parts, user input is only used in predefined places and is escaped.

5.6 Privacy

One of the requirements (section 2.5.2.1) is that only public data should be indexed. This makes access control unnecessary.

5.7 Cluster

Because HS needs a search inside user created data that does not grow rapidly and can reindex it in acceptable time, single primary shard will suffice. To scale number of queries per second, replica shards can always be added.

5.8 Conclusion

Now that I designed how data should be stored and searched, it needs to be implemented in accordance with non-functional requirements defined in section 2.5.2.

Project	
project_id	keyword
updated_at	date
created_at	date
current_code	keyword
code	keyword
amount_required	double
amount_gathered	double
leader_name	text
leader_position	text
direct_support_account_name	text
direct_support_account_number	keyword
direct_support_account_iban	keyword
direct_support_account_bic	keyword
direct_support_nank	text
location_name	text
location_address	text
location_gps	geo_point
has_stone	boolean
country_id	keyword
continent_id	keyword
category_Id	keyword
en_name	text
en_summary	text
en_impact	text
en_challenge	text
en_country_name	text
en_category_name	text
en_continent_name	text
en_timeline_text	text
cs_name	text
cs_summary	text
cs_impact	text
cs_challenge	text
cs_country_name	text
cs_category_name	text
cs_continent_name	text
cs_timeline_text	text

Figure 5.1: Denormalized data of HS project

Chapter **6**

Realization

In this chapter I describe how I implemented my design from previous chapter.

6.1 Analyzers

Because HS must search multilingual data, all these localized texts need to be properly analyzed in order to provide effective search.

6.1.1 AnalyzerInterface

To provide an easy way to add and change custom analyzers, I created a mechanic that loads analyzer definitions form PHP classes and creates them in Elasticsearch. I created AnalyzerInterface, which defines methods for getting information necessary to create Elasticsearch analyzers and few additional methods that can answer questions like what fields and languages this analyzer analyzes. Based on this information, the search module can automatically add new analyzers to Elasticsearch and dynamically create new analyzed fields.

Method *getName* is used to identify analyzer, which is saved under this name inside Elasticsearch.

Method *supportsLanguage* is used to identify which language this analyzer can analyze. For example, the default analyzer which is language-unaware returns always true. Method supportsField answers the question whether this analyzer can analyze a specific field.

I can add different analyzer for name, description and organization website by simply creating a new class defining the analyzer. Adding new analyzers requires only creating new class implementing the AnalyzerInterface and registering the class in Symfony Service Container[31] with appropriate tag.

[31]`https://symfony.com/doc/3.4/service_container.html`

The mapping class then receives all analyzers in an array as its parameter and defines analyzed fields in the mapping. This allows for adding new analyzers without changing existing code.

6.1.2 Mapping generation

Elasticsearch mapping is generated automatically from a list of searchable fields (in MySQL) and a list of analyzers. It first creates mapping for all un-analyzed fields, this is hardcoded inside the mapping generator. Then it takes a list of languages, a list of analyzers and a list of fields and creates mapping for analyzed fields. For analyzed fields it uses following algorithm:

- Iterate over languages
- Iterate over fields
- Find all analyzers that work with current field and language
- Create new field analyzed by this analyzer,
  naming it *languageName_analyzerName_fieldName*

6.2 Indexing

In this section I describe how I implemented loading project data from MySQL and indexing them in Elasticsearch.

6.2.1 Fields

Instead of duplicating strings containing field names in multiple places, I created *final class Fields* which contains constants with field names. Constants are public and a static method *getAll* uses reflection to get all constants and returns them inside an array. I didn't use enumeration type because PHP does not have one.

6.2.2 Project Document class

Maps project properties to a document mapping inside Elasticsearch using field name and information from analyzers.

For example two analyzers *A* and *B*, analyzing field *name*, one for Czech and the other for English. *Project Document* then takes value from field *name* and indexes it in *cs_a_name* and *en_b_name*.

When a *Project Document* is filled with all the data, it is converted to a JSON document, which can be indexed inside Elasticsearch.

6.2.3 Bulk indexing

Elasticsearch offers a special bulk operations for inserting, updating and deleting documents [72]. Main advantage of bulk operations is speed, which can speed up indexing throughput significantly compared to sending requests one by one.

When indexing, the indexer creates a batch of *Project Documents* with data, converts them to JSON and indexes them using the Batch API.

6.2.4 Project indexer class

Project indexer is the heart of indexing, where all parts come together. It is responsible for loading data from MySQL, denormalizes them by adding data to *Project Document* and then indexes them in batches into Elasticsearch.

My first version of the indexer used naive approach, using the ORM's *getAll* method, however when indexing large, randomly generated data, it slowly consumed all memory, including the swap and eventually crashed.

This was because ORMs are meant for transactional operations, loading an entity from database, performing an action and saving the changes. Because it hydrates objects and keeps copies to perform dirty checking[32], using ORM consumes a lot of CPU and memory. Because loading data to be indexed in different system is one-way process,I do not need the ORM's features for tracking changes. So I loaded the data using the DQL into arrays instead of objects. Due to the way MySQL handles chained one to many joins, a query that fetches everything at once duplicates a lot of data, returning lots of rows for one translation. This uses a lot of memory in the PHP process and makes putting the data inside *Project Document* harder. To speed things upI first load category, country and continent translations, store them inside an associative array and perform join in the application level. It is more efficient, because these entities are fixed, their number does not grow during application lifetime, but the number of projects does.

By doing so, each project translation is a row of SQL results, ordered by project ID. NowI just go over the rows, combining all that have same project ID. I can also load rows in batches, which saves memory and is necessary for working with larger data.

[32] `https://www.doctrine-project.org/projects/doctrine-orm/en/2.6/reference/change-tracking-policies.html`

This implementation made the indexing not crash, use a lot less memory, approximately the size of a batch multiplied by the size of the largest project and it consumes less CPU time.

6.2.5 Index command

To execute the process of indexing I created a Command Line Interface (CLI) application using the Symfony framework Console component[33]. It is a library that helps with creating CLI applications, by providing helpers for parsing input and formatting output.

6.3 Query construction

I created domain specific query builder, searching for a matching value is done by calling a method searchBy with parameters field name and the searched value. This method can be called for multiple times for the same field, in which case it searches for both values, adding their score together. It saves the field names and values and when converting this object into Elasticsearch query it calls corresponding transformer for each field.

6.3.1 Transformers

Each field has a transformer, it is a class that knows how a field should be queried and generates corresponding Elasticsearch query for searched values. Each transformer knows how its query is important and boosts it accordingly. In addition, transformers also boost languages the user understands.

6.3.2 Search service

Search service encapsulates Elasticsearch client from the rest of the search module. It takes HS search query, converts it to Elasticsearch query and performs search using the Elasticsearch client.

6.3.3 Search facade

Search facade is an intermediary between user input and search service. Controller uses Symfony Forms[34] and parses HTTP request into a Data Transfer Object (DTO), which is then passed to the search facade. Search facade constructs a query based on data present in DTO. It then calls search service, passing it the constructed query and returns search results. By using DTO with facade pattern I decoupled user interface from query construction.

[33] https://symfony.com/doc/3.4/components/console.html
[34] https://symfony.com/doc/3.4/forms.html

6.3.4 Pagination

HS uses KnP Components Pager[35], which is a library for pagination. Old search passed a Doctrine query object to the Pager and it, based on currently viewed page, sets offset, executes the query and offers the results to template to iterate over.

KnP Pager is designed to be extended. When query should be paginated, it dispatches an event with a query, limit and offset. Listeners waiting for this event then check whether they know how to paginate a query of its type and return items if they can. I implemented new listener which can paginate HS search query object, using the search service. By extending pagination I was allowed to keep search results template intact.

Inside the paginator I am catching exception in case Elasticsearch index does not exist, log it, and return zero results. When no match was found template shows message no matching projects found and displays random projects.

6.3.5 Tuning relevance

When I finished a working prototype of search, I tuned the importance of queried fields. I started by using priority designed in section 4.8. Then I selected few similar projects and selected words from their fields. Then I started searching using combinations of these words, paying attention to results, making sure no field was too important to skew results, adjusting field boosts accordingly, until I was satisfied with the result ordering for all my test queries.

These boosts, or even entire query, is likely to be further tuned, when real users start searching and giving feedback.

6.4 Tools

In this section I describe what tools I used during the development to create, explore, verify or debug parts of my solution.

6.4.1 Git

Git[36] is an open source distributed version control system [73]. HS uses it to version its source code. During the development phase I created a branch for my new search module, wrote code in iterative fashion, *committing*[37] smaller

[35] `https://github.com/KnpLabs/knp-components`
[36] `https://git-scm.com`
[37] `https://git-scm.com/docs/git-commit`

units of code. I used *rebase*[38] to reapply my commits on top of newest changes from upstream. Before merging to upstream, I am going to *squash*[39] my commits into one logical commit without history of unsuccessful prototypes.

6.4.2 Kibana

Kibana[40] is "*an open source analytics and visualization platform designed to work with Elasticsearch. Kibana can be used to search, view, and interact with data stored in Elasticsearch indices.*" [74].

It can display mapping information, which I used to check my generated mapping was correct. Its discover[41] feature allows browsing indexed data and running queries.

It offers developer console[42] which sends requests directly to Elasticsearch and displays formatted results. This console is useful in combination with several Elasticsearch API endpoints that help with debugging queries and text analysis.

The *_analyze* endpoint takes as an input text and analyzer name and returns analyzed tokens [75]. I used this feature to debug my custom analyzers.

The *_explain* endpoint takes a query and returns information about document score for each subquery [76]. I used this feature to debug my search query.

It also has special UI to profile the performance of search queries[43].

6.4.3 Xdebug

Xdebug[44] is a PHP extension which provides a debugger [77]. I used it to step through query generation when generated query wasn't what was intended.

6.4.4 Blackfire.io

To profile my solution and fix some bottlenecks during development I used Blackfire[45]. It is Performance Management Solution which can profile time, memory, and I/O of execution of PHP requests [78]. It can also display comparisons between two profiles to see changes in performance. Blackfire can be used as part of continuous integration using performance assertions.

[38] `https://git-scm.com/docs/git-rebase`
[39] `https://git-scm.com/book/en/v2/Git-Tools-Rewriting-History#_squashing`
[40] `https://www.elastic.co/products/kibana`
[41] `https://www.elastic.co/guide/en/kibana/6.2/discover.html`
[42] `https://www.elastic.co/guide/en/kibana/6.2/console-kibana.html`
[43] `https://www.elastic.co/guide/en/kibana/6.2/xpack-profiler.html`
[44] `https://xdebug.org`
[45] `https://blackfire.io`

6.5 Conclusion

In this chapter I explained how I implemented my design. Implemented solution at the time of writing this chapter contains over 3.5 thousand lines of code. (not counting configuration, shell and other non-PHP files) In the next chapter I am going to describe how I ensured this code is in good shape.

Chapter 7

Testing

In this chapter I describe what I did to ensure quality and correctness of my implementation.

7.1 Manual testing

I went through existing projects and searched for different forms of important words I found in them. I searched with different forms of words, words in different order, with different ASCII characters, with typos and with synonyms. I also searched for metadata like countries or categories. By doing so I verified my implementation was able to search in real-world data and it fulfils all functional requirements specified in section 2.5.1.

I also my search implementation to a HS representative, who was satisfied with the search results and accepted my implementation.

7.2 Integration tests

HS uses Codeception[46] framework for integration tests. It is Behavior Driven Development (BDD) framework which allows writing unit, functional, integration, and acceptance tests in a single style [79].

Integration tests are intended to test several parts of the system together, for example a feature without UI.

[46] `https://codeception.com`

I used integration tests to verify that following sequence works:

- Creating new query object
- Submitting the query object to search service
- Converting query to valid Elasticsearch request
- Elasticsearch executes search query
- Search service correctly interprets response
- Returned items contain expected results

In BDD tests are written in the form of interactions, "I do X" to perform an action and "I see Z" to verify the action was successful. An example would be an action "I navigate to https://google.com" and a check "I see response code was 200" followed by "I see Google search field".

To be able to test search in BDD way, I created two methods that are used to perform a search and check results, *seeInResult* and *dontSeeInResult*. The former is used to verify that query matches certain results. The latter is used to verify that query does not match certain results. An example Codeception test (*Cest*) can be seen in listing 2.

For this to work, there must be corresponding testing data indexed inside Elasticsearch. I achieved this by loading a sql file with data into MySQL and then running the index command, if there is a bug in indexing, these tests also have a chance of finding it.

Elasticsearch provides its own methods to evaluate the quality of search results via Ranking Evaluation API. It offers API for measuring *precision at K (P@k)* which measures the number of relevant results in top K documents, *mean reciprocal rank* to measure how high is the first relevant item in the search results page and *discounted cumulative gain* which evaluates positions of all relevant items in top K results [80]. In contrast to my solution, these metrics evaluate ranking, while my solution only evaluates matching (yes/no).

Although more advanced, basing tests on these metrics would be unstable, because any change in boosting/scoring might change the results of these methods.

This API is "*experimental and may be changed or removed completely in a future release*" [80], because of that and because of potential test instability, I decided not to use it.

```php
<?php

class FullTextCest
{

  // Set things up

  public function SummaryTest(search $I): void
  {
    $this->text->searchBy(Fields::SUMMARY, 'Eco librari at indonezia');

    $relevantProjectIds = [
      'project_A_id',
      'project_B_id',
    ];

    $irrelevantProjectIds = [
      'project_C_id',
    ];

    $I->seeInResult($this->query, $relevantProjectIds);
    $I->dontSeeInResult($this->query, $irrelevantProjectIds);
  }
}
```

Listing 2: Example of an integration test

7.3 Static analysis

To ensure code quality I used static analysis tools.

7.3.1 PHP Parallel Lint

PHP Parallel Lint[47] scans PHP files and validates PHP language syntax. This is useful because PHP is interpreted language and therefore any mistake in syntax would otherwise be found during interpretation.

Syntax checking is a very basic static analysis, but because it is very fast and errors it finds are very severe, it is a good first step before running any other static analysis tools or tests.

[47] https://github.com/JakubOnderka/PHP-Parallel-Lint

7.3.2 PHPStan

PHP is dynamic, weakly typed language, but it has made possible to type function and method parameters and return values. Because it is interpreted language, any type mismatch produces error during interpretation. One of PHPStan's[48] main features is type checking without running the code. It combines knowledge from function definitions and phpdoc to infer type of a variable and then checks whether operations performed with the variable are possible with its type.

7.3.3 PHPMD

PHP Mess Detector[49] is a tool that checks for code smells using predefined rules. Typical rules are function length, cyclomatic complexity and NPath complexity. It can also detect dead code, which is a code that will never be run, for example, after a return statement in a function. Based on its output,I refactored code that was too long or complicated.

7.4 Monitoring

Monitoring serves as a window inside a running system. For example, it can tell us how hardware resources are utilized or how often errors occur. This information is crucial to verifying all parts of system are running, finding bottlenecks or analyzing issues occurring in production environment.

7.4.1 Prometheus

Prometheus[50] is an open source monitoring system and time series database [81]. Prometheus offers query language specialized for time series data.

It can also use linear regression to answer questions like "based on past activity, how full will the hard drive be in an hour" [82]. Because it is prediction of state an hour in future, we can assume we have an hour to act.

In comparison, waiting until a disk usage hits certain value does not tell us how much time we have left, because it doesn't take into account how fast is the disk being filled.

[48] `https://github.com/phpstan/phpstan`
[49] `https://phpmd.org`
[50] `https://prometheus.io`

Prometheus scrapes data from exporters, which are small services that expose metrics via HTTP. It is the job of an exporter to gather data about a service. For example, Elasticsearch exporter calls Elasticsearch API to gather information about the cluster and then exposes aggregated data to Prometheus.

7.4.2 Metrics

Now that I described Prometheus, it is time for me to describe metrics I monitor.

- CPU
- Disk
- Ram
- JVM memory
- JVM processes states
- JVM garbage collection

7.4.3 Grafana

Grafana[51] is tool for querying, visualizing and alerting on metrics [83]. It supports graphing time series data from various sources like MySQL, Graphite, Elasticsearch and Prometheus. It is used across many industries, for example, used by CERN, PayPal or Energy Weather [84].

I used Grafana to visualize metrics about my system to understand what is going on and identify bottlenecks. In fig. 7.1 can be seen the CPU usage during indexing. Because PHP Elasticsearch client I use sends request synchronously[52], PHP (purple) and Elasticsearch (yellow) take turns. Each yellow spike means a batch of projects was sent to Elasticsearch be indexed and the PHP indexer is waiting for HTTP response.

7.5 Conclusion

By testing described in this chapter I verified my solution fulfils functional requirements and my code is in good condition.

[51] https://grafana.com
[52] https://github.com/ruflin/Elastica/issues/1156

Figure 7.1: Graph of CPU usage during indexing

Chapter 8

Possibilities

While researching search technologies and techniques I learned about features that were out of scope for this thesis, but could be used to further improve search in the future.

8.1 Analytics

Websites routinely use different analytical tools, like Google Analytics[53] to measure user interactions with website to identify parts to further improve. Without measuring first it is very hard to evaluate impact of changes. Similarly a search module can also measure certain metrics that help improve it.

For example we can measure user interactions:

- What the users search for and how many results we offer - what is important to them
- Zero result searches - for what searches we have no results for
- Position of items they open - are the results at the top satisfying
- Paging - how deep in result pages the users go
- What projects they found using search they donated to
- Query rewrites - when user searches for something and then dissatisfied with results rewrites the query

Based on these metrics we can further tune the search module or even shape content by purposefully onboarding new projects. It is important when implementing analytics to pay attention to user's privacy and laws like GDPR.

[53] https://www.google.com/analytics/#?modal_active=none

8.1.1 AB testing

To measure how much of an improvement are changes to search module, we need to measure changes in metrics described in section 8.1. For this purpose there is AB testing, which lets a portion of users use new search and other portion of users the older version of search. Then we need to compare metrics and conclude whether our changes are improving search or making it worse.

8.1.2 User signals

Analytics can also be used to directly improve scoring. For example, if majority of users searching for "tiger" always open result on a third place, it is likely better result then the preceding two, so we can boost it to first position.

8.2 UI

Quality of search results are dependent on the query. It is desirable for users to write a good search query in order to find what they are looking for. In order to achieve that there are several techniques that help users improve their query.

8.2.1 Autocomplete

Use words or phrases in our index to offer the user autocomplete as he types. When user uses the autocomplete, it ensures he is searching for things we already know we have and therefore we have some results for it.

8.2.2 Autoloading results

Loading results asynchronously while the users type. This allows them to see what the results are in near-realtime, without waiting for the page to reload.

8.3 Rerank

Elasticsearch offers feature called rerank, which takes a window of N top results and runs different search query only on them. This allows us to run a lot more time consuming query only on items we already know are relevant, computing more precise score. When using rerank it is best to design the first query for recall with only simple scoring and the second solely for scoring.

8.4 Quality rank

We can boost projects that we see as better quality using score function (section 4.8.2.4). Score function is custom script that is executed for each matching document and can access it's fields, to save time it is a good practice to run score function in the rerank phase described in section 8.3.

For example, we might want to boost projects that are updated frequently, publish status updates using timeline often or contain multiple higher resolution images.

It is necessary however to not overdo this feature in order for the results to be still be relevant to the user query. It is not a good idea to put perfectly fitting result to the last place, just because it has low resolution images. To make sure this does not happen, we can set-up *max_boost*.

8.5 Preprocess query

We can pre-process user's query and derive multiple queries matching more results.

8.5.1 Taxonomies

Human languages have taxonomies, in sense a hierarchy of words. This is in a sense similar to synonyms, but is more advanced, because taxonomies can be nested. For example, "natural disasters" include "floods", "wildfires", "tsunamis", "tornadoes" and "hurricanes". So when user searches for "natural disasters", apart from results containing matching words, items matching any specific natural disaster are also relevant. Or in reverse scenario searching for "tsunami" might also return other items matching "natural disasters".

To keep the user happy it is important to put results for his original query first and results for derived queries second.

8.5.2 Translation

It might happen that user submits a query in language that has no matching document. We can use machine translation and search in English language, which is required for all projects.

8.6 Similarity tuning

Some fields are shorter and more significant than others, it is possible to tune scoring by choosing different similarity algorithm and/or tuning its parameters. For example, BM25 can have adjusted saturation (k1 parameter) [85]. This can be useful for title where word saturation should be much steeper, possibly instant, than in a description.

8.7 In-place update of donations

When project receives a donation Elasticsearch needs to index changes. Currently entire document is fetched from MySQL and indexed again, deleting the old document with same id. For updates like numeric increments or addition Elasticsearch offers in place update. In place update does atomic delete and insert, but the computation and retrying happens on server, which decreases the amount of data loaded from database and sent over network.

8.8 Pagination without offset

When searching using $from$ and $size$, each time $from$ is increased (user navigates to another page of results) a search for $from+size$ items is performed and kept in memory, but only $size$ items are returned. To save resources Elasticsearch offers additional paging techniques like "search_after", which does not keep results not returned to user in memory [86].

8.9 Cache

To speed search up we can cache search results in Redis. So when another user searches using same query, we can return cached items. This might be particularly useful when some project becomes very popular.

Conclusion

I successfully implemented new search module which gives better results than the previous one and makes the stakeholders satisfied.[54] My solution fulfils both functional and non-functional requirements specified in section 2.5.

Search now uses language analyzers and can match non-exact word matches which improved recall significantly. Enumerable fields like categories, countries and continents have their names indexed as well as their IDs to allow fulltext search as well as filtering. For example, searching "czech education" will match a project that is in country "Czech Republic" and has category "education" even if the project description does not contain any matching words. Search is now able to handle synonyms, for example, searching "Czechia" and "Czech Republic" matches same results, while previously searching "Czechia" returned zero results.

All changes were done while keeping the user interface untouched. New search module allows for new fields and analyzers to be easily added without changing existing code.

I identified several features that can be further implemented to improve search module and described them in chapter 8.

While working on this thesis I used information gained in various university courses. The most relevant ones were BI-BIG[55] where I learned about distributed databases and fulltext search and BI-VWM[56].

[54] New search module is not in production yet, it is waiting to be merged and deployed.
[55] `http://bk.fit.cvut.cz/cz/predmety/00/00/00/00/00/00/03/09/40/p3094006.html`
[56] `http://bk.fit.cvut.cz/cz/predmety/00/00/00/00/00/00/01/12/39/p1123906.html`

Bibliography

1. HELPSTONE. *HelpStone website - How HelpStone works* [online] [visited on 2018-04-23]. Available from: https://www.helpstone.org/en/about/how-helpstone-works.
2. BRADNER, Scott. *RFC 2119* [online]. 1997 [visited on 2018-04-23]. Available from: https://www.ietf.org/rfc/rfc2119.txt.
3. *Redis website - Introduction* [online] [visited on 2018-04-23]. Available from: https://redis.io/topics/introduction.
4. UNITED NATIONS. *The United Nations Terminology Database - Czech Republic* [online] [visited on 2018-04-23]. Available from: https://unterm.un.org/UNTERM/Display/Record/UNHQ/NA/4275087d-4018-4082-899d-95f37efeda65.
5. NASA. *What is the difference between a tornado and a hurricane?* [online] [visited on 2018-04-24]. Available from: https://pmm.nasa.gov/resources/faq/what-difference-between-tornado-and-hurricane.
6. CORPORATION, Oracle. *MySQL documentation - Natural Language Full-Text Searches* [online] [visited on 2018-05-11]. Available from: https://dev.mysql.com/doc/refman/5.7/en/fulltext-natural-language.html.
7. CORPORATION, Oracle. *MySQL documentation - Boolean Full-Text Searches* [online] [visited on 2018-05-11]. Available from: https://dev.mysql.com/doc/refman/5.7/en/fulltext-boolean.html.
8. THE APACHE SOFTWARE FOUNDATION. *Apache Lucene Core* [online] [visited on 2018-04-24]. Available from: https://lucene.apache.org/core/.
9. BRASETVIK, Alex. *Elasticsearch from the Bottom Up, Part 1* [online]. 2014 [visited on 2018-04-24]. Available from: https://www.elastic.co/blog/found-elasticsearch-top-down.

10. BRASETVIK, Alex. *Elasticsearch from the Bottom Up, Part 1* [online]. 2013 [visited on 2018-04-24]. Available from: https://www.elastic.co/blog/found-elasticsearch-from-the-bottom-up.
11. ELASTIC. *Elasticsearch - Getting Started* [online] [visited on 2018-04-23]. Available from: https://www.elastic.co/guide/en/elasticsearch/reference/6.2/getting-started.html.
12. THE APACHE SOFTWARE FOUNDATION. *Elasticsearch licence file* [online] [visited on 2018-04-24]. Available from: https://github.com/elastic/elasticsearch/blob/v6.2.3/LICENSE.txt.
13. ELASTIC. *Elastic - Subscriptions* [online] [visited on 2018-04-23]. Available from: https://www.elastic.co/subscriptions.
14. ELASTIC. *Elasticsearch - Language Analyzers* [online] [visited on 2018-05-11]. Available from: https://www.elastic.co/guide/en/elasticsearch/reference/current/analysis-lang-analyzer.html.
15. ELASTIC. *Elasticsearch - Query DSL* [online] [visited on 2018-05-11]. Available from: https://www.elastic.co/guide/en/elasticsearch/reference/current/query-dsl.html.
16. ELASTIC. *Elasticsearch - Install Elasticsearch with Docker* [online] [visited on 2018-05-14]. Available from: https://www.elastic.co/guide/en/elasticsearch/reference/current/docker.html.
17. THOUGHTWORKS, Inc. *ThoughtWorks Technology Radar - Elasticsearch* [online] [visited on 2018-04-23]. Available from: https://www.thoughtworks.com/radar/platforms/elastic-search.
18. *Elasticsearch - Use case GitHub* [online] [visited on 2018-04-23]. Available from: https://www.elastic.co/use-cases/github.
19. WIKIMEDIA FOUNDATION, Inc. *Wikimedia moving to Elasticsearch* [online]. 2014 [visited on 2018-04-23]. Available from: https://blog.wikimedia.org/2014/01/06/wikimedia-moving-to-elasticsearch/.
20. SVINGEN, Boerge. *All the Data That's Fit to Find: Search @ The New York Times* [online]. 2016 [visited on 2018-04-23]. Available from: https://www.elastic.co/elasticon/conf/2016/sf/all-the-data-thats-fit-to-find-search-at-the-new-york-times.
21. THE APACHE SOFTWARE FOUNDATION. *Apache Solr website* [online] [visited on 2018-04-23]. Available from: http://lucene.apache.org/solr/.
22. THE APACHE SOFTWARE FOUNDATION. *Apache Solr Reference - Language Analysis* [online] [visited on 2018-05-11]. Available from: https://lucene.apache.org/solr/guide/7_3/language-analysis.html.

23. THE APACHE SOFTWARE FOUNDATION. *Apache Solr Reference - The Standard Query Parser* [online] [visited on 2018-05-11]. Available from: https://lucene.apache.org/solr/guide/7_3/the-standard-query-parser.html#the-standard-query-parser.

24. THE APACHE SOFTWARE FOUNDATION. *Apache Solr Reference - The DisMax Query Parser* [online] [visited on 2018-05-11]. Available from: https://lucene.apache.org/solr/guide/7_3/the-dismax-query-parser.html#the-dismax-query-parser.

25. THE APACHE SOFTWARE FOUNDATION. *Apache Solr Reference - The Extended DisMax Query Parser* [online] [visited on 2018-05-11]. Available from: https://lucene.apache.org/solr/guide/7_3/the-extended-dismax-query-parser.html.

26. *Docker Hub - Solr* [online] [visited on 2018-05-14]. Available from: https://hub.docker.com/_/solr/.

27. THE APACHE SOFTWARE FOUNDATION. *Solr powered* [online] [visited on 2018-04-23]. Available from: https://wiki.apache.org/solr/PublicServers.

28. ELASTIC. *Elasticsearch guide - Inverted Index* [online] [visited on 2018-05-03]. Available from: https://www.elastic.co/guide/en/elasticsearch/guide/2.x/inverted-index.html.

29. ELASTIC. *Elasticsearch guide- Pluggable Similarity Algorithms* [online] [visited on 2018-05-10]. Available from: https://www.elastic.co/guide/en/elasticsearch/guide/current/pluggable-similarites.html.

30. CONNELLY, Shane. *Practical BM25 - Part 2: The BM25 Algorithm and its Variables* [online] [visited on 2018-05-10]. Available from: https://www.elastic.co/blog/practical-bm25-part-2-the-bm25-algorithm-and-its-variables.

31. ELASTIC. *Elasticsearch guide - The Unit of Scale* [online] [visited on 2018-05-03]. Available from: https://www.elastic.co/guide/en/elasticsearch/guide/2.x/shard-scale.html.

32. ELASTIC. *Elasticsearch guide - Replica Shards* [online] [visited on 2018-05-03]. Available from: https://www.elastic.co/guide/en/elasticsearch/guide/2.x/replica-shards.html.

33. ELASTIC. *Elasticsearch - Mapping* [online] [visited on 2018-05-03]. Available from: https://www.elastic.co/guide/en/elasticsearch/reference/current/mapping.html.

34. ELASTIC. *Elasticsearch - Text datatype* [online] [visited on 2018-05-03]. Available from: https://www.elastic.co/guide/en/elasticsearch/reference/6.2/text.html.

35. ELASTIC. *Elasticsearch - Keyword datatype* [online] [visited on 2018-05-03]. Available from: https://www.elastic.co/guide/en/elasticsearch/reference/6.2/keyword.html.
36. ELASTIC. *Elasticsearch - Number datatype* [online] [visited on 2018-05-03]. Available from: https://www.elastic.co/guide/en/elasticsearch/reference/6.2/number.html.
37. ELASTIC. *Elasticsearch - Boolean datatype* [online] [visited on 2018-05-03]. Available from: https://www.elastic.co/guide/en/elasticsearch/reference/6.2/boolean.html.
38. ELASTIC. *Elasticsearch - Date datatype* [online] [visited on 2018-05-03]. Available from: https://www.elastic.co/guide/en/elasticsearch/reference/6.2/date.html.
39. ELASTIC. *Elasticsearch - Geo-point datatype* [online] [visited on 2018-05-03]. Available from: https://www.elastic.co/guide/en/elasticsearch/reference/6.2/geo-point.html.
40. ELASTIC. *Elasticsearch - Object datatype* [online] [visited on 2018-05-03]. Available from: https://www.elastic.co/guide/en/elasticsearch/reference/6.2/object.html.
41. ELASTIC. *Elasticsearch - Array datatype* [online] [visited on 2018-05-03]. Available from: https://www.elastic.co/guide/en/elasticsearch/reference/6.2/array.html.
42. ELASTIC. *Elasticsearch - Nested datatype* [online] [visited on 2018-05-03]. Available from: https://www.elastic.co/guide/en/elasticsearch/reference/6.2/nested.html.
43. ELASTIC. *Elasticsearch - Analysis* [online] [visited on 2018-05-03]. Available from: https://www.elastic.co/guide/en/elasticsearch/reference/6.2/analysis.html.
44. ELASTIC. *Elasticsearch - Anatomy of an analyzer* [online] [visited on 2018-05-03]. Available from: https://www.elastic.co/guide/en/elasticsearch/reference/6.2/analyzer-anatomy.html.
45. CHOLAKIAN, Andrew. *All About Analyzers, Part One* [online] [visited on 2018-04-25]. Available from: https://www.elastic.co/blog/found-text-analysis-part-1.
46. ELASTIC. *Elasticsearch - Character Filters* [online] [visited on 2018-05-10]. Available from: https://www.elastic.co/guide/en/elasticsearch/reference/current/analysis-charfilters.html.
47. ELASTIC. *Elastic - HTML filter* [online] [visited on 2018-04-26]. Available from: https://www.elastic.co/guide/en/elasticsearch/reference/6.2/analysis-htmlstrip-charfilter.html.

48. ELASTIC. *Elasticsearch - Tokenizers* [online] [visited on 2018-05-10]. Available from: https://www.elastic.co/guide/en/elasticsearch/reference/6.2/analysis-tokenizers.html.
49. ELASTIC. *Elasticsearch - Token Filters* [online] [visited on 2018-05-10]. Available from: https://www.elastic.co/guide/en/elasticsearch/reference/6.2/analysis-tokenfilters.html.
50. ELASTIC. *Elastic - Lowercase filter* [online] [visited on 2018-04-26]. Available from: https://www.elastic.co/guide/en/elasticsearch/reference/6.2/analysis-lowercase-tokenfilter.html.
51. ELASTIC. *Elastic - Asciifolding filter* [online] [visited on 2018-04-26]. Available from: https://www.elastic.co/guide/en/elasticsearch/reference/6.2/analysis-asciifolding-tokenfilter.html.
52. ELASTIC. *Elastic - Stemmer filter* [online] [visited on 2018-04-26]. Available from: https://www.elastic.co/guide/en/elasticsearch/reference/6.2/analysis-stemmer-tokenfilter.html.
53. ELASTIC. *Elastic - Hunspell filter* [online] [visited on 2018-04-26]. Available from: https://www.elastic.co/guide/en/elasticsearch/reference/6.2/analysis-hunspell-tokenfilter.html.
54. ELASTIC. *Elastic - Keword repeat filter* [online] [visited on 2018-04-26]. Available from: https://www.elastic.co/guide/en/elasticsearch/reference/6.2/analysis-keyword-repeat-tokenfilter.html.
55. ELASTIC. *Elastic - Unique filter* [online] [visited on 2018-04-26]. Available from: https://www.elastic.co/guide/en/elasticsearch/reference/6.2/analysis-unique-tokenfilter.html.
56. ELASTIC. *Elastic - Synonym filter* [online] [visited on 2018-04-26]. Available from: https://www.elastic.co/guide/en/elasticsearch/reference/6.2/analysis-synonym-tokenfilter.html.
57. ELASTIC. *Elastic - Stop filter* [online] [visited on 2018-05-14]. Available from: https://www.elastic.co/guide/en/elasticsearch/reference/6.2/analysis-stop-tokenfilter.html.
58. ELASTIC. *Elasticsearch - Match Query* [online] [visited on 2018-05-03]. Available from: https://www.elastic.co/guide/en/elasticsearch/reference/6.2/query-dsl-match-query.html.
59. ELASTIC. *Elasticsearch - Match Phrase Query* [online] [visited on 2018-05-03]. Available from: https://www.elastic.co/guide/en/elasticsearch/reference/6.2/query-dsl-match-query-phrase.html.
60. ELASTIC. *Elasticsearch - Phrase Matching* [online] [visited on 2018-05-03]. Available from: https://www.elastic.co/guide/en/elasticsearch/guide/current/phrase-matching.html.

61. ELASTIC. *Elasticsearch - Term query* [online] [visited on 2018-05-03]. Available from: https://www.elastic.co/guide/en/elasticsearch/reference/6.2/query-dsl-term-query.html.

62. ELASTIC. *Elasticsearch guide - Prefix query* [online] [visited on 2018-05-03]. Available from: https://www.elastic.co/guide/en/elasticsearch/reference/6.2/query-dsl-prefix-query.html.

63. ELASTIC. *Elasticsearch - Range query* [online] [visited on 2018-05-03]. Available from: https://www.elastic.co/guide/en/elasticsearch/reference/6.2/query-dsl-range-query.html.

64. ELASTIC. *Elasticsearch - Ranges* [online] [visited on 2018-05-03]. Available from: https://www.elastic.co/guide/en/elasticsearch/guide/current/_ranges.html.

65. ELASTIC. *Elastic - Function Score Query* [online] [visited on 2018-04-26]. Available from: https://www.elastic.co/guide/en/elasticsearch/reference/6.2/query-dsl-function-score-query.html.

66. ELASTIC. *Elasticsearch - Coumpound queries* [online] [visited on 2018-05-03]. Available from: https://www.elastic.co/guide/en/elasticsearch/reference/6.2/compound-queries.html.

67. ELASTIC. *Elasticsearch - Dis Max Query* [online] [visited on 2018-05-03]. Available from: https://www.elastic.co/guide/en/elasticsearch/reference/6.2/query-dsl-dis-max-query.html.

68. ELASTIC. *Elasticsearch - Bool Query* [online] [visited on 2018-05-03]. Available from: https://www.elastic.co/guide/en/elasticsearch/reference/6.2/query-dsl-bool-query.html.

69. ELASTIC. *Elasticsearch - Query and filter context* [online] [visited on 2018-05-03]. Available from: https://www.elastic.co/guide/en/elasticsearch/reference/current/query-filter-context.html.

70. ELASTIC. *Elasticsearch - Optimistic Concurrency Control* [online] [visited on 2018-05-03]. Available from: https://www.elastic.co/guide/en/elasticsearch/guide/current/optimistic-concurrency-control.html.

71. PAQUETTE, Mike. *Elasticsearch - Protecting Against Attacks that Hold Your Data for Ransom* [online] [visited on 2018-05-10]. Available from: https://www.elastic.co/blog/protecting-against-attacks-that-hold-your-data-for-ransom.

72. ELASTIC. *Elasticsearch - Bulk API* [online] [visited on 2018-05-11]. Available from: https://www.elastic.co/guide/en/elasticsearch/reference/current/docs-bulk.html.

73. *Git* [online] [visited on 2018-05-14]. Available from: https://git-scm.com.

74. ELASTIC. *Kibana - Introduction* [online] [visited on 2018-05-11]. Available from: https://www.elastic.co/guide/en/kibana/6.2/introduction.html.

75. ELASTIC. *Elasticsearch - Analyze API* [online] [visited on 2018-05-11]. Available from: https://www.elastic.co/guide/en/elasticsearch/reference/current/indices-analyze.html.

76. ELASTIC. *Elasticsearch - Explain API* [online] [visited on 2018-05-11]. Available from: https://www.elastic.co/guide/en/elasticsearch/reference/current/search-explain.html.

77. RETHANS, Derick. *Xdebug website* [online] [visited on 2018-05-11]. Available from: https://xdebug.org.

78. SAS, Blackfire. *Blackfire - Features* [online] [visited on 2018-05-11]. Available from: https://blackfire.io/features.

79. CODECEPTION. *Codeception - Introduction* [online] [visited on 2018-05-11]. Available from: https://codeception.com/docs/01-Introduction.

80. ELASTIC. *Elasticsearch - Ranking Evaluation API* [online] [visited on 2018-05-10]. Available from: https://www.elastic.co/guide/en/elasticsearch/reference/6.2/search-rank-eval.html.

81. PROMETHEUS. *Prometheus - Overview* [online] [visited on 2018-05-11]. Available from: https://prometheus.io/docs/introduction/overview/.

82. PROMETHEUS. *Prometheus - Functions* [online] [visited on 2018-05-11]. Available from: https://prometheus.io/docs/prometheus/latest/querying/functions/.

83. LABS, Grafana. *Grafana* [online] [visited on 2018-05-11]. Available from: https://grafana.com/grafana.

84. LABS, Grafana. *Grafana - Testimonials* [online] [visited on 2018-05-11]. Available from: https://grafana.com/grafana/testimonials.

85. ELASTIC. *Elastic - Similarity modules* [online] [visited on 2018-04-26]. Available from: https://www.elastic.co/guide/en/elasticsearch/reference/6.2/index-modules-similarity.html.

86. ELASTIC. *Elasticsearch - Search After* [online] [visited on 2018-05-03]. Available from: https://www.elastic.co/guide/en/elasticsearch/reference/6.2/search-request-search-after.html.

Appendix A

Glossary

AMQP Advanced Message Queuing Protocol

API Application Programmable Interface

BDD Behavior Driven Development

CLI Command Line Interface

Doctrine ORM Object Relation Mapper library

DQL Doctrine Query Language

DTO Data Transfer Object

Elastic The company behind Elasticsearch

Elasticsearch OpenSource software for text search

HS HelpStone `https://helpstone.org`

MySQL Relational database

NGO Non-profit, non-governmental Organization

ORM Object Relation Mapper

OSS Open Source Software

PHP PHP Hypertext Preprocessor

SQL Simple Query Language

UI User Interface

UID Unique identifier

Appendix **B**

Contents of enclosed CD

```
/ ..................................................... root directory
├── thesis.pdf .......................... the thesis text in PDF format
└── thesis_source/ ....... directory with the thesis text in source format
```