FACULTY
OF MATHEMATICS
AND PHYSICS
Charles University

MASTER THESIS

Bc. Adam Hornáček

Suggester implementation for the OpenGrok search engine

Network and Labs Management Center

Supervisor of the master thesis:	Mgr. Vladimír Kotal
Study programme:	Computer Science
Study branch:	Software and Data Engineering

Prague 2018

I declare that I carried out this master thesis independently, and only with the cited sources, literature and other professional sources.

I understand that my work relates to the rights and obligations under the Act No. 121/2000 Sb., the Copyright Act, as amended, in particular the fact that the Charles University has the right to conclude a license agreement on the use of this work as a school work pursuant to Section 60 subsection 1 of the Copyright Act.

In ..... date ..... signature of the author

I would like to thank my supervisor, Mgr. Vladimír Kotal, for the opportunity to work on the OpenGrok project and for his guidance and useful advice. I also thank my family for their support during my studies.

Title: Suggester implementation for the OpenGrok search engine

Author: Bc. Adam Hornáček

Institute: Network and Labs Management Center

Supervisor: Mgr. Vladimír Kotal, Network and Labs Management Center

Abstract: The suggester functionality is an important feature of modern search engines. The aim of the thesis is to implement it for the OpenGrok project. The OpenGrok search engine is based on Apache Lucene and supports its query syntax. Presented suggester implementation supports this query syntax and provides suggestions not only for prefixes but also for wildcards, regular expressions, or phrases. The implementation also takes into account the possibility of grouping queries. That means, if one query is already specified and user is typing another query, then the first query will restrict the suggestions for the second query. The promotion of specific suggestions is based on the underlying Lucene index data structure and previous searches of the users.

Keywords: autocomplete suggester OpenGrok index Lucene

Contents

Introduction

Internet has become an unavoidable component of everyday life of billions of people. Among other things, it serves as a source of knowledge which is available literally at our fingertips. However, it is not possible for a single person to filter or find relevant information from such a huge dataset. As a result, many search engines emerged which try to provide this functionality. Sometimes, users do not know for what they are exactly looking or the search engine can predict for what the user could be looking based on some criteria. These criteria consist mainly of what the user already typed into the system or the user's search history. Nevertheless; many others may be integrated as well, e.g. the system can better predict search based on some personal information about the user (age, gender, etc.) or searches of other users.

Suggester, or autocomplete, functionality tries to achieve this deficiency by giving the user the possibility to choose from a handful of choices it deemed the most attractive for the user. Therefore, suggester implementations were added to many search engines (e.g. Google[1] – it should be noted that OpenGrok is a specialized search engine and should not be compared with Google directly). However, OpenGrok[2] search engine does not contain any suggester implementation.

0.1 Target of the thesis

The thesis aims to fulfill the following targets:

- Integrate suggester functionality into OpenGrok.
- Suggestions provided by the suggester should be meaningful and helpful.
- The suggester should provide support for rich configuration.
- Evaluate the impact of the suggester on the system running the OpenGrok instance.

0.2 Structure

Chapter 1 **OpenGrok** provides a quick overview of OpenGrok project. What tools it uses and what makes the OpenGrok project extraordinary.

Chapter 2 **Analysis** discusses problems that arose during the implementation process and how they were resolved.

Chapter 3 **User Documentation** contains pictures of the added functionality and explains how it should be used.

Chapter 4 **Program Documentation** gives an explanation of implementation specific properties of the work. In other words, how exactly was the new content integrated into the OpenGrok project.

[1] `http://google.com`
[2] `http://opengrok.org`

Chapter 5 **Impact on the OpenGrok Project** provides detailed description, comparison and evaluation of the functionality that affected the OpenGrok project.

1. OpenGrok

1.1 Overview

OpenGrok is an open-source search engine written in Java[1] freely available at `https://github.com/oracle/opengrok` under CDDL license[2]. However, OpenGrok is different to most of the search engines, purpose of which is to traverse web and find the most relevant webpages. OpenGrok provides searching across codebases. For instance, if you would like to know which files are using *mmap* function in *Linux kernel*[3] then OpenGrok is the tool for that purpose. It understands a variety of programming languages, e.g. Java, C, C#, JavaScript and many others. The other most notable features of OpenGrok are:

- Support for multiple projects. **Project** is a directory containing source files. Most commonly, it is a directory containing the source files for one software project; thus, the name.
- Support for authentication and authorization (Tulinger [2017]). For instance, by using LDAP [4].
- Support for multiple version control systems[5], e.g. git, mercurial, etc.
  - Support for searching in the repository history.
  - Possibility to annotate a file.
  - Possibility to see the history for a specific file.
- Support for cross-referencing. In the context of OpenGrok, abbreviation **xref** is used.
- Support for navigation in source files. For instance, by providing list of methods.
- Syntax highlighting of source files.

1.2 Modules

OpenGrok is split into multiple modules:

- jrcs – Java library which provides support for RCS[6] version control system.
- Indexer – described in the Section 1.2.1.
- Web – described in the Section 1.2.2.

[1] `https://en.wikipedia.org/wiki/Java_(programming_language)`
[2] `https://en.wikipedia.org/wiki/Common_Development_and_Distribution_License`
[3] `https://en.wikipedia.org/wiki/Linux_kernel`
[4] `https://en.wikipedia.org/wiki/Lightweight_Directory_Access_Protocol`
[5] `https://en.wikipedia.org/wiki/Version_control`
[6] `https://en.wikipedia.org/wiki/Revision_Control_System`

- plugins – contains authorization plugins.
- distribution – does not contain any code and serves for combining all of the above into one distribution unit.

Visualization of these modules and how they are dependent on each other can be seen in Figure 1.1.

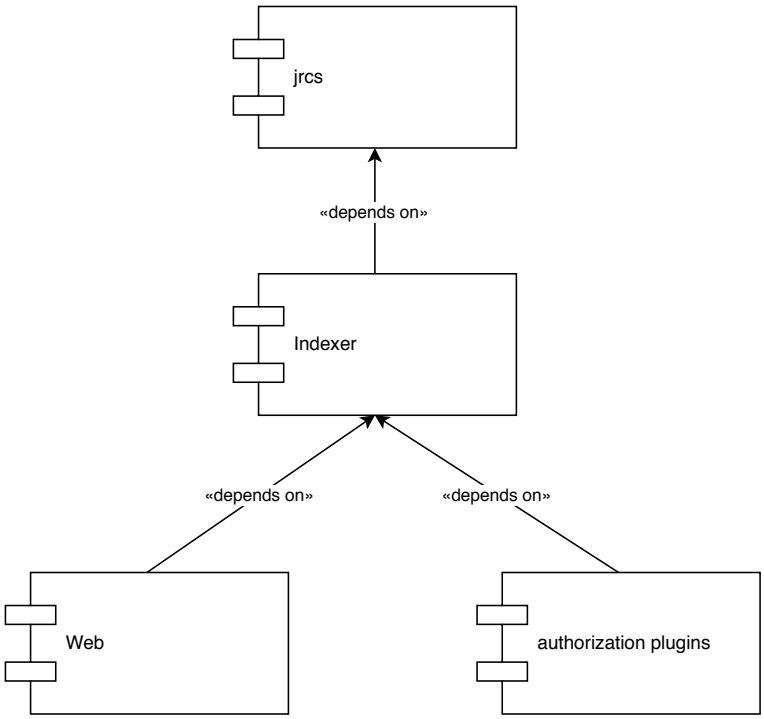

Figure 1.1: OpenGrok modules

Multiple-module support in one project is achieved by Maven[7] build tool. There is one main *pom.xml* in the root of the project which aggregates all the modules.

1.2.1 Indexer

Indexer is a submodule, main purpose of which is for each project to:

- Generate the main lookup index. Detailed description in the Section 1.2.1.
- Generate the history index for repositories. Detailed description in the Section 1.2.1.
- Generate xref. Detailed description in the Section 1.2.1.

Index

OpenGrok heavily relies on Apache Lucene[8] [McCandless et al., 2010], a search engine library written in Java. Every source file is considered to be a *document*.

[7] `https://maven.apache.org`
[8] `https://lucene.apache.org`

Every *document* has multiple *fields*s which contain text data. This text data is then tokenized by specific rules (e.g. remove stop words, stemming, etc.) into tokens. In the Lucene context *term* is a pair of *field* and *token*. Lucene uses *reverse document index*[9], i.e. for each term a collection of document identifiers is stored in which this term occurs. This form of data representation is very efficient for document lookup.

Tokenization OpenGrok uses Ctags[10] to detect definitions. OpenGrok uses custom analyzers for symbol tokenization. Each analyzer is specific to some file format, e.g. Java or JavaScript.

History

OpenGrok uses the history index for quick access to history information of every file in a project. OpenGrok first discovers what kind of version control system the project is using and then executes appropriate commands to retrieve the project's history (e.g. some form of *git log*). Since executing these commands is not very efficient (new process has to be created), OpenGrok does this at the indexing phase to provide better performance later. Serialized form of Java object is stored for each source file of the project to represent its history. To be precise, OpenGrok supports multiple repositories in one project and history files are generated for the specific repository to which the file belongs.

Xref

Main purpose of the OpenGrok is to find relevant documents based on the search query. After the successful search, the most obvious route of action is to look at the found documents. However, analyzing the document once again, adding cross-reference links, performing syntax highlighting and other operations can be very time consuming. Therefore, *xref* directory contains pre-generated HTML[11] versions of every source file which already contain the aforementioned information.

1.2.2 Web

The Section 1.2.1 describes how the data are processed to create the index. However, these data need to be presented to the end user. The Web submodule serves exactly this purpose. It is distributed as a WAR[12] which can be used by many servlet containers[13]. OpenGrok correctly runs in Apache Tomcat[14]; nevertheless, there is no Tomcat vendor lock-in; therefore, other servlet containers should work properly as well.

[9] `https://en.wikipedia.org/wiki/Inverted_index`
[10] `https://en.wikipedia.org/wiki/Ctags`
[11] `https://en.wikipedia.org/wiki/HTML`
[12] `https://en.wikipedia.org/wiki/WAR_(file_format)`
[13] `https://en.wikipedia.org/wiki/Web_container`
[14] `http://tomcat.apache.org`

Configuration

Since OpenGrok has two modules (Indexer and Web) which can be run separately, it needs a way to pass relevant information between those. However, it might be possible that Indexer finishes its work after a long period of time (in some cases it might take days) and the Web module is not reachable. Therefore, to not lose all of the information needed by the Web module, the Indexer stores it in a configuration file. It is stored as an XML[15] encoded class which needs to be read by Web on its startup. If the Web module does not find the configuration at the expected place (can be configured in *web.xml*) then an error is shown on the main page and the usage is not possible. Of course, there are ways to change the configuration of the running OpenGrok application via the REST API [16] described in the Section 1.2.2.

REST API

OpenGrok provides REST API support. This is a relatively new feature. Before that, OpenGrok had known a concept of *Messages* – custom serialization of Java objects passed to the Web application via a custom port. So far, most of the REST API calls can be only made from the machine on which the OpenGrok runs. This is mainly because these REST API calls are meant as a means of communication between the Indexer and Web application or for administrators maintaining the OpenGrok instance. More information can be found in [Kotal and Hornáček].

1.3 Usage

The main purpose of the OpenGrok project is to provide ability to search codebases of software projects. This is done mainly through its web interface. The main OpenGrok page can be seen in the Figure 1.2.

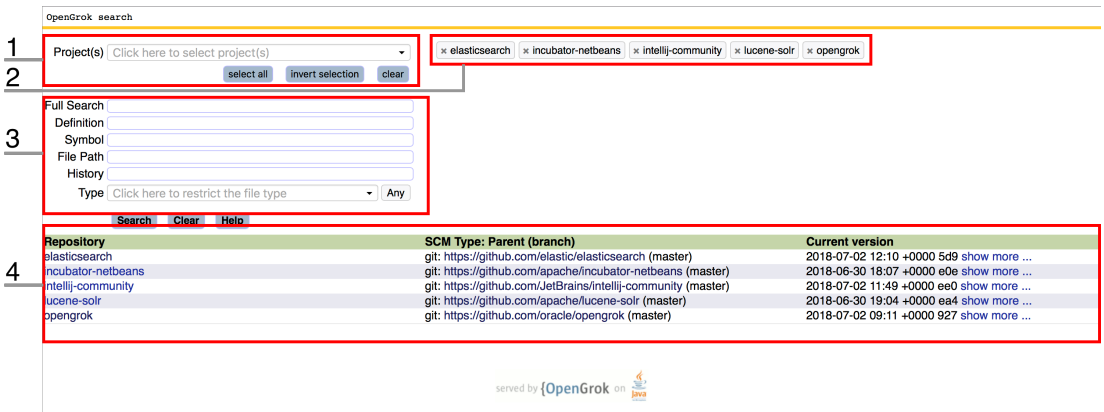

Figure 1.2: Main OpenGrok page

The numbers in the Figure 1.2 signify:

1. Field where projects can be selected.

[15] https://en.wikipedia.org/wiki/XML
[16] https://en.wikipedia.org/wiki/Representational_state_transfer

2. Selected projects.

3. Search fields.

   - **Full search** – specifies the text the document should contain. Represents *full* field in underlying Lucene implementation. It is possible to specify the search criteria for other fields by using the Lucene's {*field*}:{*value*} format.
   - **Definition** – specifies the definition of a symbol the document should contain. Represents *defs* field in the underlying Lucene implementation.
   - **Symbol** – specifies the symbol the document should contain. Represents *refs* field in the underlying Lucene implementation.
   - **File Path** – specifies the path the document should possess. Represents *path* field in the underlying Lucene implementation.
   - **History** – specifies the history the document should possess. Represents *hist* field in the underlying Lucene implementation.
   - **Type** – restricts the search for documents of the type. Represents *type* field in the underlying Lucene implementation.

4. Repository list.

It is possible to add values to multiple search fields to make the search more specific. Furthermore, any query written in Lucene syntax can be typed into the search fields.

More detailed information with examples can be found on the */help.jsp* web page of the running OpenGrok Web application instance.

1.3.1 Navigating Results

OpenGrok presents search results similarly to what can be seen in Figure 1.3.

In the Figure 1.3 numbers represent:

1. Results are paged as is a custom in many applications.

2. Directory where the result documents are located.

3. Document that satisfies the search criteria.

4. Line number and content of the line where the match was found.

1.3.2 Traversing Source Code

OpenGrok presents files' content similarly to modern IDEs[17]. This can be noticed in the Figure 1.4 where numbers represent:

1. Line numbers.

2. File content.

[17] `https://en.wikipedia.org/wiki/Integrated_development_environment`

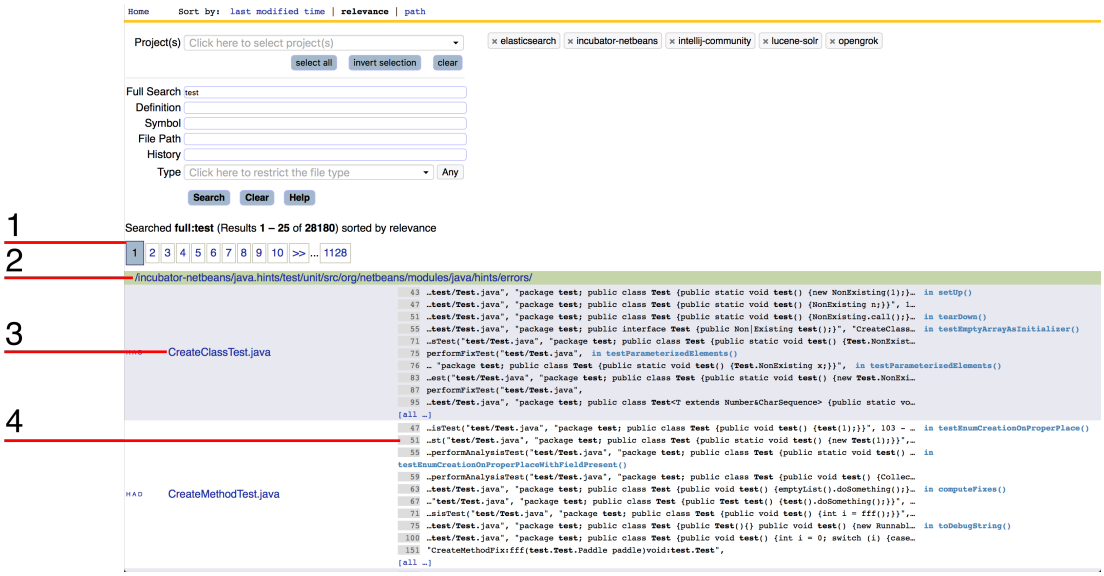

Figure 1.3: OpenGrok search results

3. Current code scope of the content that is displayed.

4. Window with quick navigation links across the source file.

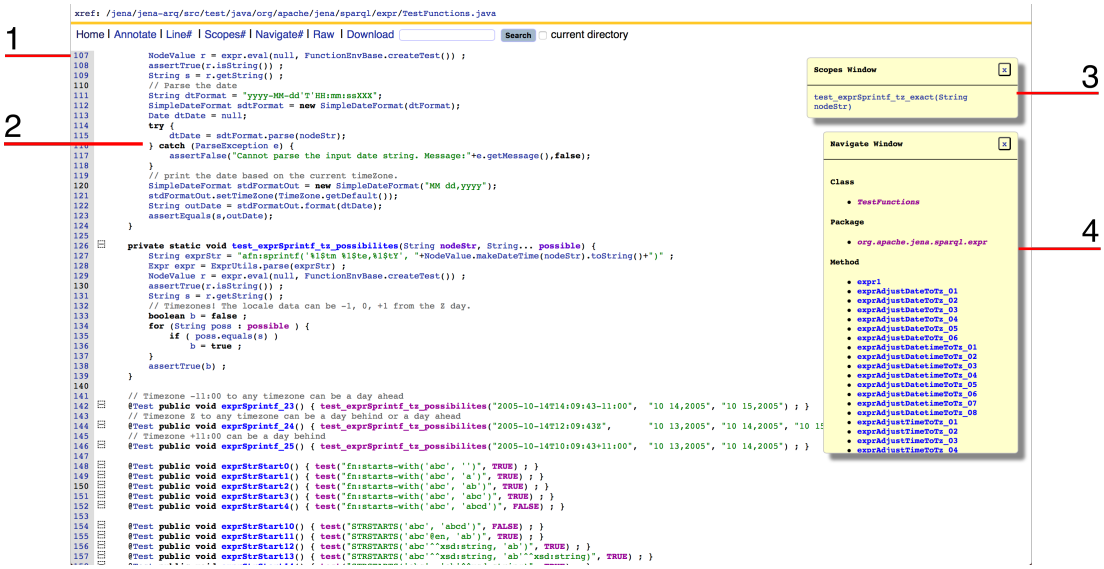

Figure 1.4: OpenGrok file content

1.4 Similar Tools

It is hard to compare OpenGrok to other similar tools because all of these are complex and each of them provides a unique feel and distinct set of problems. Very often, these problems are solved in a different manner. Therefore, the list only mentions if these tools contain suggester functionality. Still, this might not be the best indicator because they might provide different solutions, e.g. instant

search which might render the suggester a redundant asset. The list of similar tools:

- **SourceGraph** `https://about.sourcegraph.com` – SourceGraph has the instant search feature. While typing some text into the search field, it immediately displays the files in which the text has been found. The Figure 1.5 shows how this feature looks like.

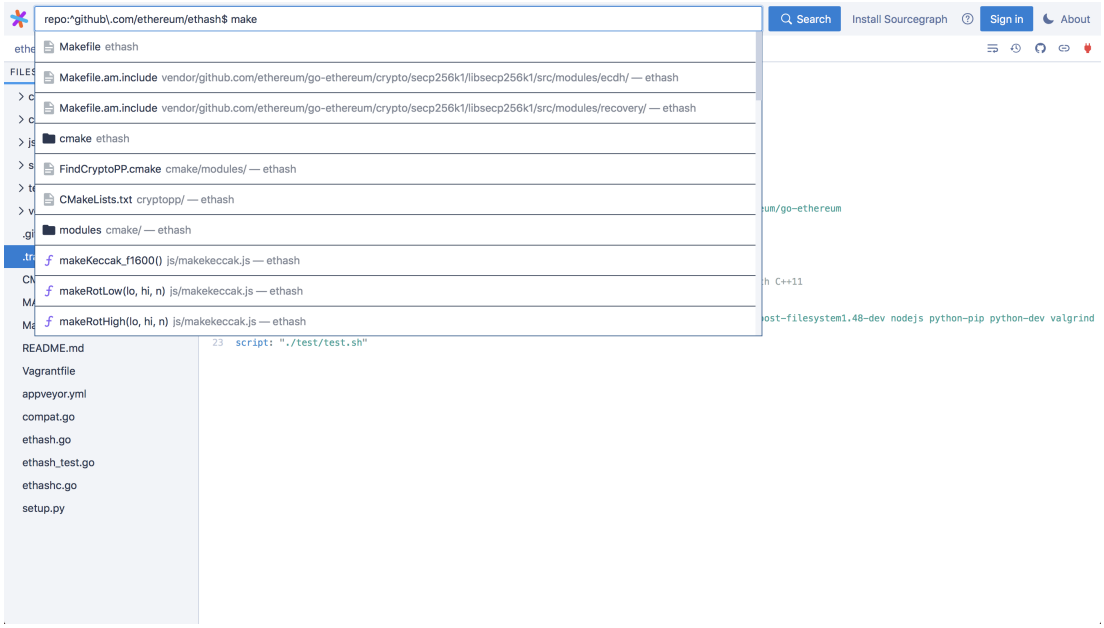

Figure 1.5: SourceGraph instant search feature

- **searchcode** `https://searchcode.com/` – it seems that searchcode does not support neither autocomplete nor instant search feature.

- **livegrep** `https://livegrep.com/` – supports instant search. Immediately shows files and their contents where the text has been found. The look of this application can be seen in Figure 1.6.

- **Debian Code Search** – does not seem to have any suggester support. It was created as a bachelor thesis [Stapelberg, 2012]. It searches all the open source projects which are included in the Debian archive. The search is based on the article [Cox, 2012] which explains how Google Code Search worked.

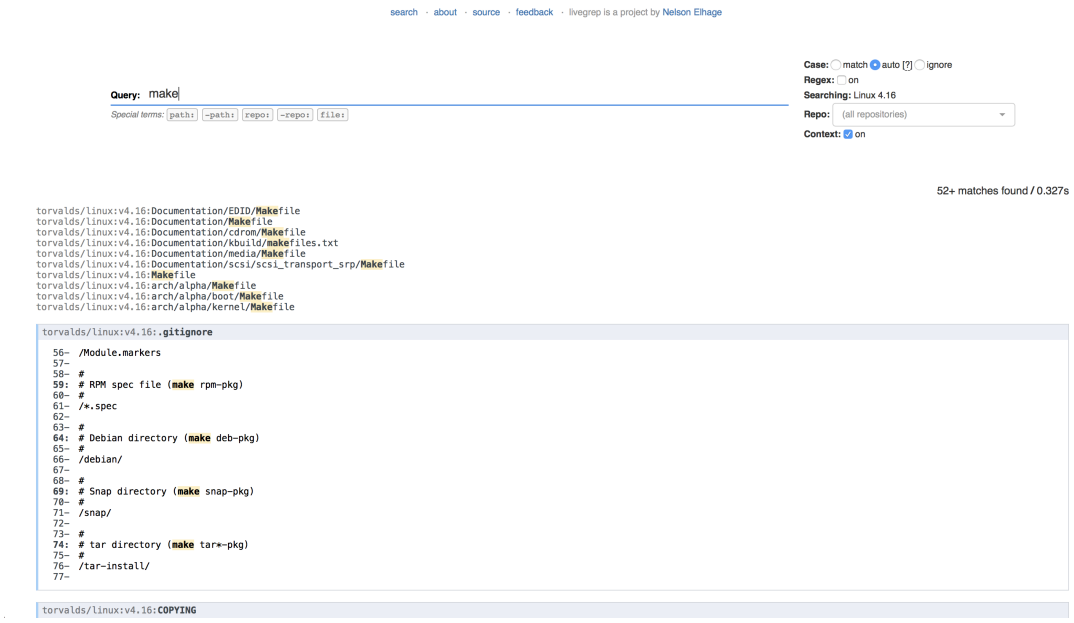

Figure 1.6: livegrep instant search feature

2. Analysis

This chapter is dedicated to the problems that arose during the work on the project and explains the major decisions that were made. The chapter is divided into the following sections:

- 2.1 **General Architecture** – explains the chosen suggester architecture and how it could be combined into the overall OpenGrok architecture.
- 2.2 **OpenGrok Modifications** – describes the major modifications that had to be made in OpenGrok code to enable suggester functionality.
- 2.3 **Suggester** – provides detailed explanation of how the suggester functionality was implemented.

2.1 General Architecture

One of the most important aspects of software products is their architecture. If the architecture does not take into account some quality attributes then they are very hard to implement in the underlying code. When designing a high level architecture, there were multiple solutions from which to choose:

- **Implement suggester in the Web module** – probably the easiest solution. There would be no inter-tier boundaries; therefore, no need to define public API [1]. Also, there would be no need to add any other configurations to the build scripts. However, this seems like a step towards a monolithic application [2] which has proven to be harder to maintain in the long run.
- **Implement suggester as a separate module** – suggester could be implemented as an additional module to the already modular architecture described in the Section 1.2. This new module could be a dependency of the Web module which would need to be modified to parse and process the data that are passed in. These data then could be passed to the new module via its public API. The result would look like shown in the Figure 2.1.
- **Implement suggester as a separate service** – this is a step towards microservices architecture [3]. The suggester would run as a separate program listening on a different port and possibly on a different machine. Therefore; the implementation could be in any programming language and not necessarily in Java. Also, this could enable easy scalability of this service, e.g. one service per project. However, the suggester would still require access to the index files.

Chosen solution – in the end, the second proposal was chosen. Implementing suggester as a separate module is a good compromise between a separate

[1] `https://en.wikipedia.org/wiki/Application_programming_interface`
[2] `https://en.wikipedia.org/wiki/Monolithic_application`
[3] `https://en.wikipedia.org/wiki/Microservices`

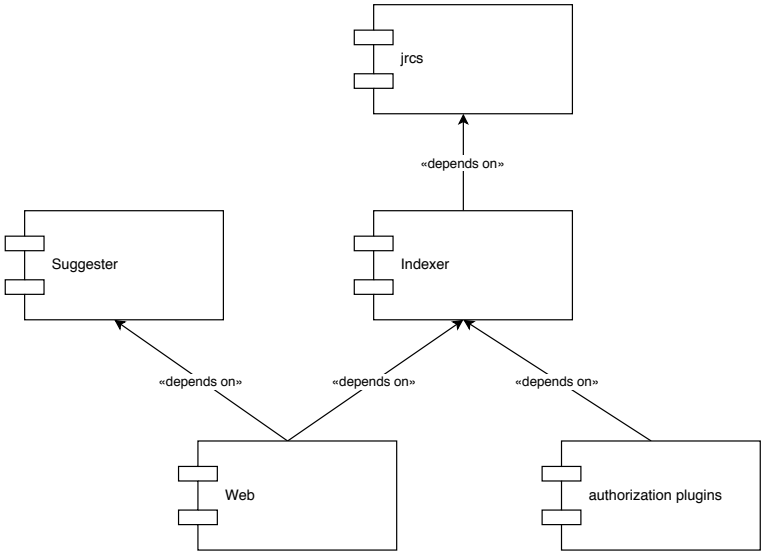

Figure 2.1: OpenGrok modules with Suggester module

service and implementing it in the Web module. The main reason for this choice was that OpenGrok does not support installation or scalability across multiple machines and these features are not planned to be implemented in the near future. Therefore, it could be hard for users to set up and configure this new service. However, the suggester module needs to have a public API and does not depend on any other OpenGrok modules. Therefore, it will not require a lot of changes to bundle it as a separate service if the need arises in the future.

2.2 OpenGrok Modifications

As discussed in the Section 2.1, the Suggester is added as a separate module which is a dependency of the OpenGrok Web module. Therefore, changes to the Web module are required. OpenGrok uses the Maven module structure. Thus, the Suggester module has to respect it as well.

2.2.1 Frontend

As described in the Section 1.2.2, OpenGrok provides a web interface. Some parts of the web need to be modified for the Suggester support. Client-side scripting is done by the use of JavaScript[4] programming language.

Configuring Suggester

OpenGrok provides REST API endpoint for querying or manipulating the configuration of the Web application. This endpoint could be used to retrieve the suggester configuration and apply it. However, this endpoint is only reachable from the targeted machine because of security reasons. Therefore, it is needed to bypass this limitation and provide one endpoint which will return only the

[4] `https://en.wikipedia.org/wiki/JavaScript`

suggester configuration. This endpoint could also filter the information which is redundant to the JavaScript implementation.

Showing the Suggestions

The Suggester needs to detect if the user pressed a key while having a specific input selected for which it is enabled. Upon detecting this change, it needs to process the data, send it to the backend part of the software, process the returned result and show it to the user. All this should be as quick as possible so the user considers it to be seamless.

In the field of search engines, speed is very important. According to [Linden], Marissa Mayer[5], former Yahoo![6] CEO[7], spoke at the 2006 Web 2.0 conference where she said that 0.5 s drop in speed reduced the traffic by 20 %.

Showing suggestions is considered to be a response to control activation. This is an indication that some control member (in our case keyboard key) has been physically activated. Based on [Miller, 1968], the response should take no more than 100 ms in order for the system to be considered unobtrusive.

Simplified picture of the steps that need to be taken is shown in the Figure 2.2.

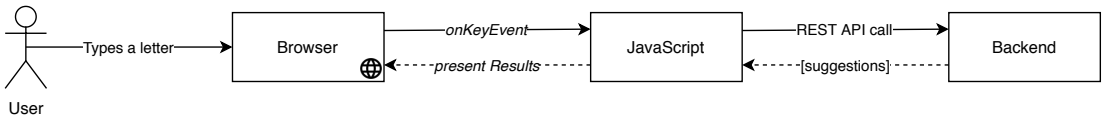

Figure 2.2: Typical suggestions workflow

There are many libraries which provide "JavaScript" functionality as shown in Figure 2.2, the most notable are:

- jQuery UI Autocomplete[8]
- Ajax Autocomplete for jQuery[9]
- MagicSuggest[10]

All of the above require jQuery[11] library which is already included in the OpenGrok. In the end, jQuery UI Autocomplete was chosen because jQuery UI was already used by OpenGrok. Although, the Autocomplete module was not included. jQuery UI Autocomplete provides an easy-to-use API and is easily customizable; these factors also contributed to the decision of picking this library.

[5] `https://en.wikipedia.org/wiki/Marissa_Mayer`
[6] `https://en.wikipedia.org/wiki/Yahoo!`
[7] `https://en.wikipedia.org/wiki/Chief_executive_officer`
[8] `https://jqueryui.com/autocomplete/`
[9] `https://www.devbridge.com/sourcery/components/jquery-autocomplete/`
[10] `http://nicolasbize.com/magicsuggest/`
[11] `http://jquery.com`

Data format As mentioned, user's input must be sent to the backend part of the software, processed and returned back. jQuery UI Autocomplete supports the JSON[12] format by default. This format is widely used and performs better than XML in this specific situation (`https://www.w3schools.com/js/js_json_xml.asp`). Also OpenGrok's REST API supports automatic conversion to JSON objects via the Jackson[13] library. Therefore, it is a solid choice.

Detecting the Token for Suggestions

As referenced in the Section 1.3, it is possible to write multiple tokens into one *input* HTML tag. Imagine the text: "car engine". With only this information the Suggester is not able to tell if it should show suggestions for the "car" prefix or "engine" prefix or something totally different. Therefore, the Suggester also requires the position of the last typed key.

The position needs to be retrieved and set by the JavaScript part of the web page. This functionality could be written in pure JavaScript; however, simple, small, and easy-to-use library *jQuery caret* (`https://github.com/acdvorak/jquery.caret`) was used instead.

2.2.2 Backend

As stated in the Section 2.2.1, the server part needs to have a service which returns results based on the input typed by the user. And as mentioned in the Section 1.2.2, the Web module is a web application compliant with the servlet API; therefore, a new servlet which listens to the user's input and returns suggestions needs to be added.

How to Implement the Suggester Servlet

The servlet represents a simple REST API endpoint because it needs to respond to HTTP GET requests and return the response in JSON. There is a lot of options how this could be achieved; however, the most notable are:

- Use default implementation by extending the *HttpServletRequest*[14] and its method *doGet()*. The main advantage of this approach is that there are no other dependencies because they are provided by the underlying servlet container. However, some boilerplate code would need to be written which could be avoided.
- Use JAX-RS API and its implementation Jersey[15]. With this approach the code can be clean and much easier to understand because of the use of provided annotations from JAX-RS API. Not to mention that Jersey provides additional features:
  - Dependency injection.

[12] `https://www.json.org`
[13] `https://github.com/FasterXML/jackson`
[14] `https://docs.oracle.com/javaee/7/api/javax/servlet/http/HttpServletRequest.html`
[15] `https://jersey.github.io`

  - Bean validation support.
  - Automatic JSON conversion using Jackson library.

Since OpenGrok Web has already REST API support and both JAX-RS API and Jersey already on the classpath, there is no dependency overhead. As a result, the second option was chosen.

Processing User Input

As referenced in the Section 1.3, OpenGrok web interface provides multiple HTML *input* fields which represent different Lucene *fields* that should be searched. Each of these input fields can contain some value. Therefore, all these values are passed to the server along with the information of the currently selected input field and the caret position in it.

Each of these fields can include any form of Lucene query, as stated in the Section 1.3. Since these queries might get very complicated, a proper parser is needed. Exactly for this purpose, OpenGrok provides modified parsers from Lucene library.

The character position is useless after the parsing is done – it is not possible to find the token it referenced anymore. Therefore, a small trick can be used. Before parsing the text data, a short, random but regarding the text unique string of characters is generated and inserted at the mentioned position. From now on, it is possible to use just this identifier since it uniquely identifies the token in which the user is currently interested. However, this presents also a small disadvantage. If the query typed by the user is not correct, then Lucene parser generates *ParseException* which contains this identifier; therefore, it might be a little harder to decode the original query from the exception's message.

Reacting to the Events

Running OpenGrok Web instance has REST API which allows the Indexer or administrators to indicate that some event occurred. More information can be found in the Section 1.2.2. Events that the Suggester should be able to react to:

- **Configuration change** – *PUT /api/v1/configuration*

  Suggester should rebuild to be in concordance with the new configuration.

- **Indication to refresh index searchers** – *PUT /api/v1/system/refresh*

  Most of the time this message indicates that the reindex of the data has been done. The Suggester should update its data to be up to date.

- **Marking project as indexed** – *PUT /api/v1/projects/{project}/indexed*

  OpenGrok can be aware of projects which are not yet indexed. It is not possible to search in these projects; as a result, there is no need for suggester to know about them. However, this changes if the project is marked as indexed. The Suggester should then build its inner data to support suggestions for the project.

- **Project delete** – *DELETE /api/v1/projects/{project}*

  Suggester should drop any data it has regarding the project.

2.3 Suggester

This section looks at the Suggester module in more detail. First of all, it describes all the queries the suggester supports. Then, the fuzzy and boost factors are discussed. After that, it discusses how to leverage the previous searches. At last, some additional properties that might be important to this module are considered.

2.3.1 Prefix Query

To start with, let's look at the simplest and at the same time probably the most used case. **Prefix query** is a special case of *Wildcard query* (described in the Section 2.3.2). It can be divided into 2 parts:

- **prefix** – text which represents the prefix of all the terms this query accepts.
- ***** – wildcard symbol which accepts any text (even zero-length text).

These parts are then combined together as: {*prefix*}*. It is probably the most frequently used case because of the suggester's nature. If the user types any simple text (even without *) then the suggester automatically appends * to the end and considers this as a prefix query to provide adequate suggestions.

Since the prefix queries will be used that often then the suggester needs to be optimized for this type of queries.

Scoring

Every term needs to have a *score* associated with it so the suggester can decide which term is better suited to appear as a suggestion. Therefore, the problem is not only to find the terms with a specific prefix but to find n terms with the best scores having the specific prefix. Let's assume that the suggester does not have any information about the previous searches. This case is discussed in the Section 2.3.10. The suggester has access to the Lucene index provided by the Indexer. Therefore; it has access to:

- **Total term frequency** T – how many times the term appears in the document set.
- **Document frequency** N – in how many documents does the term appear.
- **Number of documents** D – total number of documents in the document set.

IDF[16] At a first glance, it might seem as a good idea to use *inverse document frequency* (IDF) as a metric. The formula for computing this metric can be seen in Equation 2.1.

$$IDF(t) = \log \frac{D}{N} \tag{2.1}$$

Basically, it says how important the term is in the document set. If the term occurs in every document, then it is probably a very common word. If provided

[16] `https://en.wikipedia.org/wiki/Tf%E2%80%93idf%23Inverse_document_frequency_2`

in the query, it does not reduce the size of the result set significantly. However, this measure provides good results if the term is already supplied by the user in the query because the system can focus on the terms with the highest IDF. If the suggester were to use it, then it would suggest terms that are used the least, e.g. in the worst case only once. Therefore; this is not a good scoring option.

Document frequency Document frequency is a good indicator of how the term is used throughout the document set. However, this metric is specific to the project. Thus, it would be hard to compare values across multiple projects. The solution to the problem is normalization. The normalization is performed by dividing the document frequency of the term by the total number of documents in the document set. The result is the Equation 2.2.

$$score = \frac{N}{D} \cdot constant \tag{2.2}$$

This metric gives results which are expected of the suggester; therefore, it is the best option so far.

Possible Implementations

As mentioned before, the implementation should be fast to be able to fulfill the users' expectations. Although, at the same time, it should not waste memory resources. One data structure per Lucene field per project is needed. The OpenGrok installation can contain hundreds of projects and depending on the size of the project, one Lucene field can possibly contain tens of thousands of terms or even more (in some cases millions).

Trie[17] – sometimes called prefix tree, is the simplest structure suitable for our purposes. However, it does not support efficient lookup based on scores. It would be needed to traverse the whole subtree for a specific prefix to find the terms with the best scores.

TST[18] – (ternary search tree) is a special case of Trie. Same as Trie, it does not have any support for more efficient lookup based on scores.

FST[19] – (finite state transducer) is a finite state automaton which can produce output. A special case of FST is a weighted FST (WFST) where each transition can have an associated weight. More formally, a weighted finite-state transducer T over a semiring $\mathbb{K}$ is an 8-tuple $T = (\Sigma, \Delta, Q, I, F, E, \lambda, \rho)$ where:

- Σ is the finite input alphabet of the transducer;
- Δ is the finite output alphabet;
- Q is a finite set of states;

[17] `https://en.wikipedia.org/wiki/Trie`
[18] `https://en.wikipedia.org/wiki/Ternary_search_tree`
[19] `https://en.wikipedia.org/wiki/Finite-state_transducer`

- $I \subseteq Q$ the set of initial states;
- $F \subseteq Q$ the set of final states;
- $E \subseteq Q \times (\Sigma \cup \{\varepsilon\}) \times (\Delta \cup \{\varepsilon\}) \times \mathbb{K} \times Q$ a finite set of transitions;
- $\lambda : I \mapsto \mathbb{K}$ the initial weight function;
- $\rho : F \mapsto \mathbb{K}$ the final weight function mapping F to $\mathbb{K}$.

An overview of WFST algorithms can be found in [Mohri, 2004]. It is possible to construct a WFST which will provide a very efficient algorithm for finding the n terms with the best scores. This algorithm is explained in [Mohri and Riley, 2002]. However, this algorithm returns terms with the lowest scores and based on the chosen scoring it is needed for it to return the terms with the highest scores. This can be achieved by a neat trick; it is possible to invert the score by the Equation 2.3.

$$newScore = maxPossibleScoreValue - score \tag{2.3}$$

An efficient construction of minimal, acyclic and deterministic WFST is explained in [Mihov and Maurel, 2001].

Example – WFST with terms and weights
$CAR = 10, CARS = 15, CART = 5$ can be seen in the Figure 2.3.

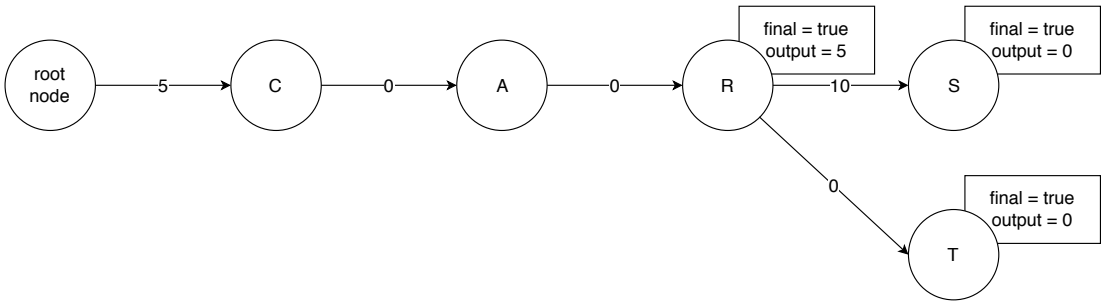

Figure 2.3: WFST example

Comparison Two datasets were used:

- **English words** – list of **370,099** words available at `https://github.com/dwyl/english-words`. It is distributed under the Unlicense license [Bendiken, 2010].
- **Linux kernel** – *full* field of indexed Linux kernel with **3,459,733** terms.

Measured was the average time of specific operations on the data structures. The description of the test cases:

- **"non" Prefix Lookup** – is a lookup of 10 words with the prefix "non" which is the most occurring prefix of length 3 in the English words dataset with 7479 occurrences. Linux kernel has only 644 occurrences. Results can be seen in the Figure 2.4.

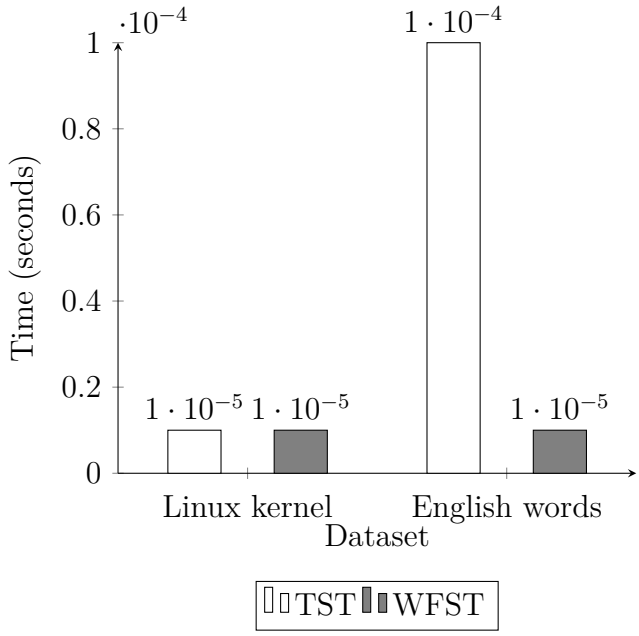

Figure 2.4: ”non” Prefix Lookup comparison

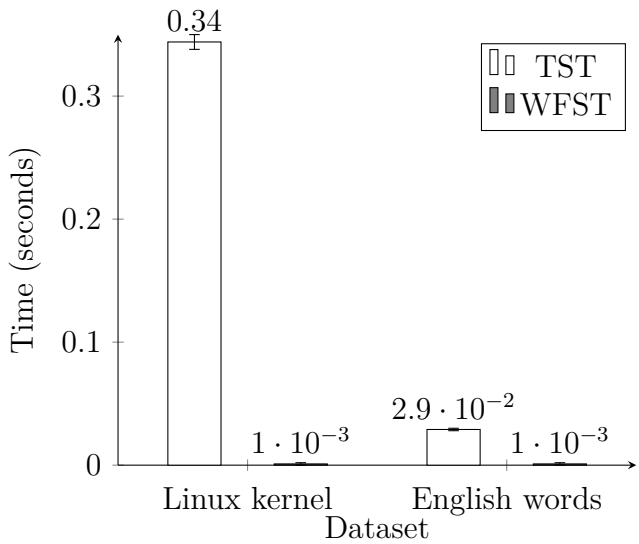

Figure 2.5: One Letter Prefix Lookup comparison

- **One Letter Prefix Lookup** – is a lookup of 10 words with the prefix of each letter from the basic alphabet. Results can be seen in the Figure 2.5.
- **Two Letter Prefix Lookup** – is a lookup of 10 words with the prefix of each combination of 2 letters from the basic alphabet. Results can be seen in the Figure 2.6.

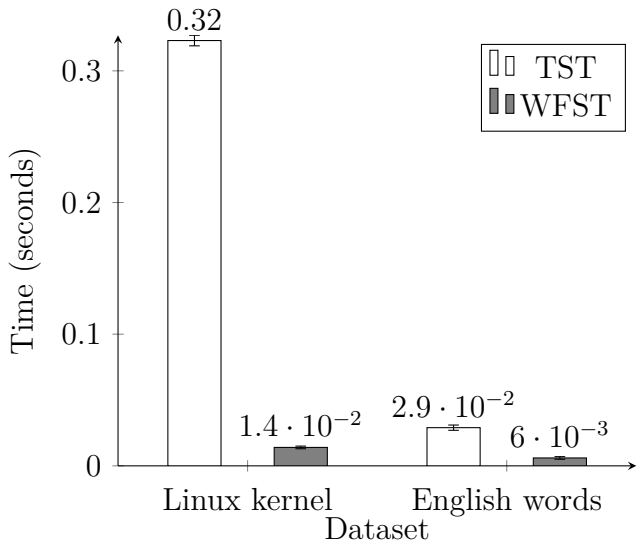

Figure 2.6: Two Letter Prefix Lookup comparison

- **Build time** – performs build of the data structure from the scratch. Results can be seen in the Figure 2.7.

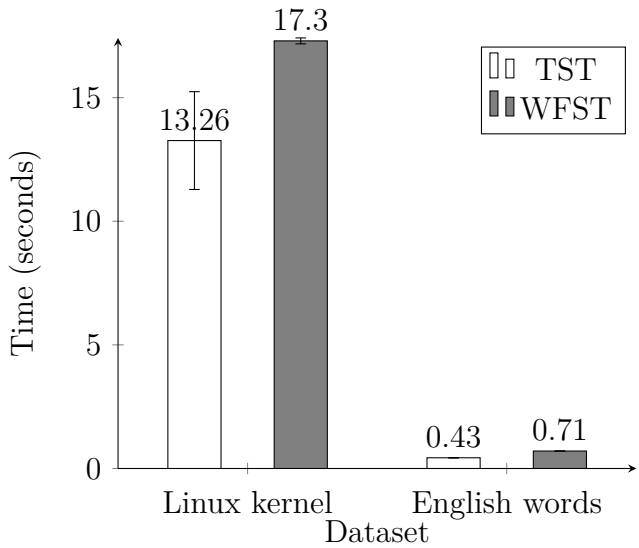

Figure 2.7: Comparison of build times

- **Memory usage** – approximate sizes the data structures use in the computer's main memory. The result can be seen in the Figure 2.8. The data was measured by calling method *ramBytesUsed()* which is a custom method commonly implemented in Lucene's data structures. The drastic difference

is mainly due to WFST implementation which was optimized to use as little memory as possible while providing a very solid performance. Some of the WFST optimizations:

- Storage in byte arrays. There is no need to keep references to the objects which take 8B on 64-bit computer but rather smaller identifiers. Also, differences in arc weights are mostly very small. The implementation leverages this and encodes integers with variable length into the mentioned byte arrays.
- Caching of the first arcs of the FST.
- Constant arc size for the arcs into certain depth which allows binary search among the arcs. (Random access is not possible because it would require more memory which would not be used.)

It can be noted that some of these optimizations may hinder the performance. However, the results show that the WFST performs well and is memory usage efficient.

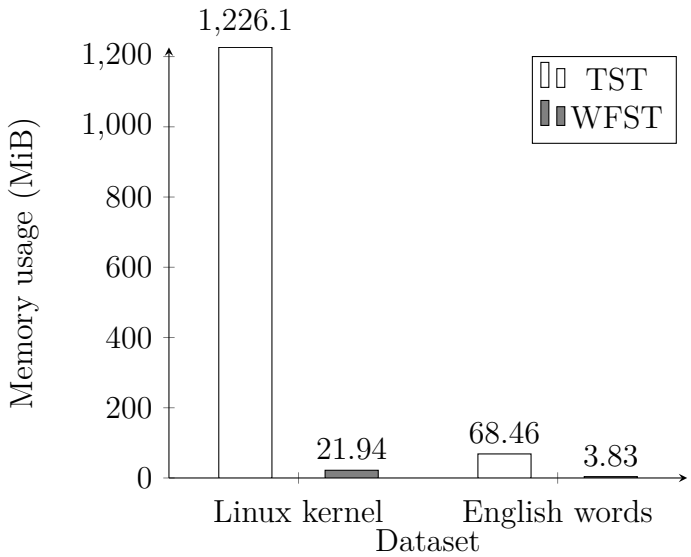

Figure 2.8: Comparison of memory usage

How to reproduce these benchmarks and description of the machine on which they were run can be found in Attachment A.

Based on the provided results, it can be noted that WFST performs much better at lookups. Mainly at lookups with a short prefix. Also, the memory usage is inarguably better in the WFST data structures. However, its build time is worse by a small margin than TST's.

OpenGrok installation can easily contain dozens of projects, each of these projects contains 5 Lucene fields (full, defs, refs, path, hist) which needs to have a suggester support. Depending on the size of the projects, TST implementation could take many GiBs of the memory which is not feasible.

To conclude, WFST is a better option in almost every aspect except the build time which is not that important because of the possibility of saving the data structures on the disk; and therefore, avoid it almost every time except when the index changes.

2.3.2 Wildcard Query

The specific case of *prefix** is covered in the previous Section 2.3.1. Therefore, all the other cases of wildcard queries will be covered in this section. The implementation of WFST cannot be used because of its nature. There is no way to efficiently search in WFST tokens for the query of type **suffix*. The required result is the same as for the prefix query: to find the terms which are accepted by the query with the top score. However, the data structure which could achieve this for generic wildcard query with the WFST performance is not known to the author. Nonetheless, the Lucene evaluation of wildcard queries could be leveraged. An automaton specific to the wildcard query is created by using the Lucene automaton implementation by replacing `?` to accept any character and `*` to accept any string. Then the terms are filtered using this automaton. The implementation is slower than WFST because all the terms need to be filtered once if they are accepted by the automaton then they need to be filtered for the second time based on their score.

Scoring

Scoring used is the same as mentioned in the Section 2.3.1.

2.3.3 Regular Expression Query

The regular expression queries can be implemented in the same way as wildcard queries (Section 2.3.2). The only difference is that the automaton is created based on the regular expression rules.

2.3.4 Phrase Query

Phrase query is a query which specifies a collection of tokens which must occur in the documents in the specified order. The tokens must be surrounded by double quotes, e.g. *"token1 token2"*. In this example, this means that *token1* must be immediately followed by *token2* in the document for it to be accepted.

Evaluation

As mentioned in the Section 1.2.1, Lucene stores data in the reverse document index. However, to be able to determine if one token is immediately followed by another token, their position needs to be stored and accessed. Lucene can be configured to store this information as well. It can easily be determined if Lucene stored this data by checking if Lucene index contains a file with a *.pos* suffix.

Evaluation of the query is straightforward. Lucene performs lookup for each token in the phrase query and intersects the results which provides all the documents which contain all of these tokens. Then it checks if the tokens are indeed next to each other and on correct positions.

Getting the Suggestions

Using shingles Shingle[20] is an n-gram of words. It would be possible to create shingles of specific count (e.g. 2 to 5 words) during the indexing phase. Each shingle would be treated as a single term. Then, it would be possible to do quick prefix lookup as described in the Section 2.3.1. However, this solution has many drawbacks:

- Another, not quite small index would have to be created.
- The suggestions would work for only the last token of the phrase query.
- The maximum number of tokens in phrase query for which the suggestions would work would be limited by the longest shingle.
- Another data structure which would contain all of the shingles for quick lookup would have to be stored in the main memory.

Using the default evaluation As explained in the Section 2.2.2, it is known which token should be considered as prefix. Therefore, this token can be omitted but the requested position of other tokens must stay unchanged. Then the same evaluation as mentioned in the Section 2.3.4 can be used. The result of the evaluation is a pair:

- Document identifier.
- Token position.

Based on this information it should be possible the gather all the possible terms and accept only those with the provided prefix.

Chosen solution The use of shingles was ruled out because of the lack of support for arbitrary number of words in the phrase query. However, it might be considered as a possible optimization if the need arises despite the added data size and memory overhead. As a result, the default evaluation was chosen.

Getting the token at specific position Based on the document identifier and token position it should be easy to get the needed term. However, Lucene does not provide any functionality to do this efficiently because the nature of the reverse document index is a complete opposite to the data structure which would provide efficient implementation of this operation. Possible solutions are:

- Implement data structure which would efficiently lookup a term based on a position and document identifier. However, the size of the data structure would not be trivial in comparison with the actual index size. Furthermore, it would slow down the indexing phase even more which might already take a few hours or even days if the projects being indexed are very huge.

[20] `https://en.wikipedia.org/wiki/W-shingling`

- Read and tokenize the document with the specified identifier. Nonetheless, this might be a slow operation if there were a lot of documents accepted by the query because of the necessary I/O[21] operations.

- Use the already existing reverse document index. It is possible to lookup all the terms based on the prefix. Then filter the terms which do not occur in the documents which were found by the phrase query and those which do not have the correct position in the document.

The first option is not feasible because of the added size to the already existing index. Comparison between the second and the third option can be seen in the Figure 2.9.

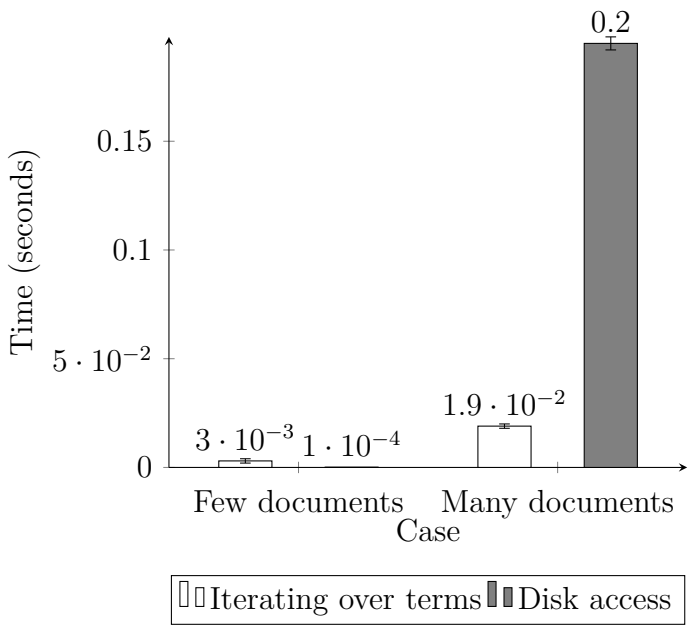

Figure 2.9: Comparison between iterating over terms and disk access for getting the term at a specific position in document

Few documents query matched only 2 documents whereas *Many documents* query matched 738 documents. The number of terms was 24355. For details about how the measurement was performed consult the Attachment A. It can be noted that for a small number of matched documents the second option is faster than the third option. However, this changes when the number of matched documents is large. Nevertheless, the second option has a few additional drawbacks:

- It needs access to OpenGrok's Indexer module to be able to tokenize the files.

- There might be problems if the source code is out of sync with the index. This happens when the project source code is updated (e.g. *git pull*) and Indexer is performing reindex.

Due to these drawbacks, the implementation where the data would be gathered by reading and tokenizing the documents was not implemented despite the

[21] `https://en.wikipedia.org/wiki/Input/output`

increased performance for a small number of documents. It might be considered as a possible extension.

Scoring

It would be possible to use the same scoring as in the Section 2.3.1. However, the scoring is not specific enough. For instance, the term could have a big document frequency but in the context of phrase query would occur only once, whereas, the term with lower document frequency could occur more times, which would make it more specific. Therefore, the scoring based on the term frequency in the context of phrase query provides better results.

2.3.5 Range Queries

Range query allows to match documents that contain terms within a certain range. It contains a lower and upper term which determine the accepted range. For instance, the query *[alfa TO beta]* accepts all terms that are lexicographically between the terms *alfa* and *beta*. Range queries have many similarities with wildcard queries (Section 2.3.2) in terms of implementation for the suggester. The main difference is in creation of the automaton. The suggestions depend on which of the terms (lower or upper) is used as a prefix.

Lower Term as a Prefix

First of all, the automaton A_{int} which represents a binary interval is created between the lower and upper term. Then, the automaton A_{prefix} which accepts all terms with prefix specified in the lower bound is needed. The result automaton is the intersection of these two, as in the Equation 2.4.

$$A_{result} = A_{int} \cap A_{prefix} \tag{2.4}$$

Upper Term as a Prefix

Similarly, the automaton A_{prefix} which accepts all terms with prefix specified in the upper bound is created. Then, another binary interval automaton A_{int} is needed; however, this one has no lower bound specified and for the upper bound the lower term of the query is used. The result automaton needs to be created as a difference between the prefix automaton and the binary interval automaton, as in the Equation 2.5.

$$A_{result} = A_{prefix} - A_{int} \tag{2.5}$$

2.3.6 Including Boolean Operators

There are multiple boolean operators which can modify the meaning of the queries.

Unary Boolean Operators

Following unary operators are supported:

- + unary operator in front of the query means that the term represented by this query must occur in the result document set.
- **-** unary operator in front of the query means that the term represented by this query cannot occur in the result document set. It is also possible to use synonyms **NOT** or **!**. However, it is not possible to use it in a query with only a single term.
- **Default** unary operator – no unary operator in front of the query means that the term represented by this query should occur in the result document set. However, the occurrence is not mandatory.

Binary Boolean Operators

At first, let's look at the simple case with only one binary boolean operator and two terms, as in the Equation 2.6.

$$term1 \diamondsuit term2 \tag{2.6}$$

If the $\diamondsuit$ is:

- **AND** – or &&, it is rewritten as in Equation 2.7.

$$+term1 \ + term2 \tag{2.7}$$

- **OR** – or ||, it is rewritten as in Equation 2.8. (Default unary operators are used.)

$$term1 \ term2 \tag{2.8}$$

Multiple binary operators It is possible to use multiple binary operators as in Equation 2.9.

$$term1 \diamondsuit term2 \diamondsuit \ldots \diamondsuit termN \tag{2.9}$$

However, there is no operator precedence (*AND* and *OR* have the same precedence). Therefore, without grouping (Section 2.3.7) it might not be clear what unary operator (default or +) will be assigned to the middle terms. Lucene parses the query sequentially from left to right and upon encountering the binary operator changes the surrounding terms. For instance, query as in Equation 2.10 will be transformed to Equation 2.11.

$$a \ \&\& \ b \ || \ c \ \&\& \ d \ || \ !e \tag{2.10}$$

$$+a \ b \ +c \ d \ -e \tag{2.11}$$

Suggester

The suggester can be interested only in unary boolean operators because binary boolean operators are transformed into unary. For simplicity, let's assume query with multiple terms (other types of queries are treated analogically). One of those terms represents a query for which the suggester needs to provide suggestions. Based on the unary operator this query has assigned, the suggester needs to:

- + – the suggested term must occur in the result set. Therefore, it is dependent on the other terms. For instance, if the query contains another term which must occur in the result set, then the suggester should only suggest the terms which are occurring in the same documents as the term. If it did not, then user could be easily confused and the result set might be empty. The solution to this problem is to:
    1. Remove the term for which the suggester needs to provide suggestions from the query.
    2. Get all the documents that satisfy the bare query.
    3. Suggest only terms which occur in at least one of the retrieved documents in step 2 and which satisfy the query removed in step 1.
- **default** – the suggested term should occur in the result set. However, if it does not, then nothing is broken. Nevertheless, suggesting terms that cannot appear in the result document set does not make much sense. Thus, it is performed as in + unary operator.
- **-** – the suggested term must not appear in the result set and as mentioned, it cannot be used in a query with a single term; therefore, its purpose is to reduce the result set. Thus, the same solution as for + needs to be performed.

If the query contains multiple terms, then more efficient data structure for autocompletion than reverse index exists [Bast et al., 2006]. Authors describe a tree-like data structure called *AUTOTREE*. In the empirical study the authors have shown that the *AUTOTREE* outperforms the basic reverse index. However, the *AUTOTREE* data structure would have to be created for every field and some minor changes would have to be made. Therefore, for now, it is considered as a possible improvement of the suggester performance.

Scoring The scoring should follow the same principle as in the Section 2.3.1 because of the consistency. However, the document frequency has to be taken from the result set of the bare query and not the whole index.

2.3.7 Grouping Queries

With boolean operators (Section 2.3.6) it is possible to create complex queries. However, it is not possible to combine queries in an easy manner. That's where grouping operators **(** and **)** have their use. For instance, it is possible to combine results of two queries as can be seen in Equation 2.12.

$$(query1) \mid\mid (query2) \tag{2.12}$$

It is also possible to use grouping operators to remove the confusion caused by the omission of precedence for the binary boolean operators. Therefore, it is possible to write queries as shown in Equation 2.13.

$$(term1\ AND\ term2)\ OR\ term3 \tag{2.13}$$

Grouping operators can be nested, thus; it is easy to create truly complex queries.

Suggester

At first, the query is parsed into multiple simple queries which are then combined together to create one complex *BooleanQuery*. It is possible to imagine the top-down decomposition into a tree-like structure. One leaf is the suggester query. This clause is marked and replaced by *MatchAllDocsQuery* which matches every document. It is possible to start at this leaf and go towards the root. On each node on the path towards the root there may be multiple clauses. The algorithm looks at all clauses and removes the clauses which have a specified *SHOULD* occurrence. The clause in which the suggester query resides is preserved by default. When the algorithm ends, the result is a stripped query which contains only queries that affect the suggestions.

Example Let's suppose that the query entered into one of the search fields has a form as in Equation 2.14. And that the suggestions are wanted for the prefix $t3$.

$$(t1\ \&\&\ ((t2\ ||\ t3)\ \&\&\ t4))\ ||\ t5 \tag{2.14}$$

The result documents with prefix $t3$ must contain the terms $t1, t4$. The result query which represents the documents that contain $t1, t4$ terms is written in Equation 2.15 ($*: *$ represents *MatchAllDocsQuery*).

$$(+t1\ +(+(+*: *)\ +t4)) \tag{2.15}$$

The tree like structure is shown in the Figure 2.10 where colors represent:

- Blue – suggestions prefix.
- Red – ignored.
- Green – accepted.

2.3.8 Dealing with Fuzzy Factor

It is possible to specify a fuzzy factor for the queries which indicates how well should the query match the documents.

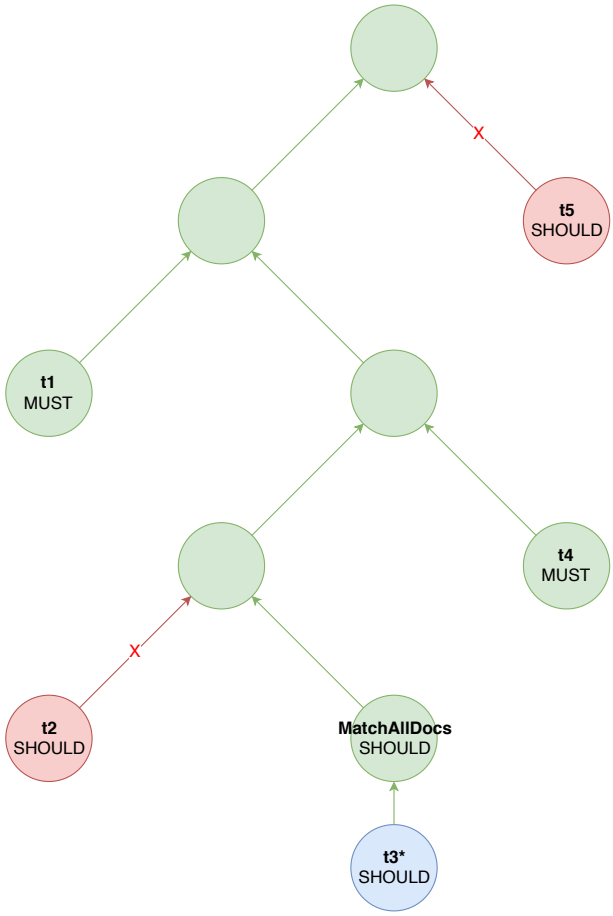

Figure 2.10: Grouping queries example tree representation

Fuzzy Search

Lucene supports fuzzy searches which are based on the Levenshtein Distance [22] algorithm. This option can be specified by typing $\sim$ after the searched term and followed by the value of the fuzzy factor. However, it can only be specified for the single term query. The Suggester supports this option. By writing the $\sim$ with the fuzzy factor after the term the suggester will suggest the terms which satisfy this condition.

Proximity Search

Proximity search is a phrase query with a specified fuzzy value. The fuzzy value represents the allowed edit distance of the tokens of the phrase query. The implementation is very similar to the one described in Section 2.3.4. However, the suggestions take the fuzzy factor into account and will contain terms that satisfy this condition.

2.3.9 Dealing with Boost Factor

Lucene supports term boosting by writing ˆ after the searched term and followed by the value of the boost factor. The default boost factor is 1. It must be positive but can be less than 1. The boost factor specifies how the term is relevant in regard to other terms. In the query specified in Equation 2.16 the term *term1* is more relevant than the term *term2*.

$$term1\hat{}3 \ \&\& \ term2 \tag{2.16}$$

It might be possible for complex queries to take this information and boost those suggested terms that are in the same document as boosted term. However, for now, the boost factor is ignored by the suggester.

2.3.10 Promoting Suggestions Using Previous Searches

The **reciprocal rank** of a query response is the multiplicative inverse of the rank of the first correct answer. That means, if the first correct answer was on the third place then reciprocal rank would be $\frac{1}{3}$.

Mean reciprocal rank[23](MRR) is defined as average of reciprocal ranks for a set of queries Q. Mathematically, it is defined as in Equation 2.17.

$$MRR = \frac{1}{|Q|} \sum_{i=1}^{|Q|} reciprocalRank_i \tag{2.17}$$

In [Bar-yossef and Kraus, 2011] multiple approaches are presented to the suggestions based on previous searches:

- **Most popular completion** – wisdom of the crowds. For some prefix the suggester suggests the terms with the same prefix that were searched the most.

[22] https://en.wikipedia.org/wiki/Levenshtein_distance
[23] https://en.wikipedia.org/wiki/Mean_reciprocal_rank

- **Nearest completion** – takes into account current context of the user that is typing the queries. The authors defined **logical search session** as an interactive process in which the user (re-)formulates queries while searching for documents satisfying a particular information need. It consists of a sequence of queries $q_1, ..., q_t (t \geq 1)$ issued by the user. The **context** of a user input x, where x is the prefix of some query q_i in the session, is the sequence of queries $q_1, ..., q_{i-1}$ preceding q_i. Authors also specified how to convert context to a vector which can be used to find the most similar query. This works for vector model based search engines. Lucene is a combination of vector and boolean model so the conversion could be applicable.

- **Hybrid completion** – if there is no or almost no context then *nearest completion* cannot predict the suggestions. Therefore, *hybrid completion* is a combination of *most popular completion* and *nearest completion*. The authors of the paper have shown in an empirical study that *hybrid completion* has better *MRR* than *most popular completion* on average.

Chosen solution – *nearest completion* and thus *hybrid completion* are very intriguing and could improve the suggestions by a big margin. However, they would need to be adapted to Lucene and the implementation might not be completely straightforward. Therefore, the basic implementation of suggester will only include the *most popular completion*. Implementation of the *nearest completion* is a very promising candidate for future extensions.

Most Popular Completion – Simple Queries

To implement most popular completion the score of the terms needs to be changed by some factor. For instance, if the term $term1$ was searched for i times then the score could be changed by a number $h(i)$ for some function $h : \mathbb{N} \mapsto \mathbb{N}$. It is also possible to take into account how recent the searches were and if there is a multiple recent searches for some term then increase its score by a higher value. For instance, this could arise in a situation if some error was found in method $m()$ of some well known library, then multiple users will probably be searching for it in a small time interval. However, the suggester will focus only on the basic implementation that considers only the term search count mainly because its simplicity. The time factor is a possible extension to an existing solution.

It is needed to have a key-value store for storing the number of searches for the terms. This store should be persistent so the data would not be lost after the restart of the application. Also this store should allow parallel access for efficiency. There are multiple options how to achieve this functionality:

- **Java Map** implementation which supports concurrent access, e.g. *ConcurrentHashMap*[24]. This map could be stored on the disk periodically to fulfill the persistency requirement. This solution has a few drawbacks:

  - Loss of recent data after restart/crash.

[24] `https://docs.oracle.com/javase/10/docs/api/java/util/concurrent/ConcurrentHashMap.html`

  - The data are held in memory. The size of the data is non-trivial, e.g. *Linux kernel* dataset contains approximately 3.5 million terms for *full* field.

- **Redis**[25] – in-memory data structure store, used as a database, cache and message broker. It has support for multiple languages and contains the needed commands (*GET, INCR*). The major drawback is that Redis would have to be installed and run separately. Although some embedded Redis solutions exist, those are meant for testing purposes. Redis also has support for persistence:
  - **RDB** persistence – performs point-in-time snapshots of the dataset at specified intervals.
  - **AOF** persistence – logs every write operation received by the server, that will be played again at server startup, reconstructing the original dataset.

- **Relational Database** – even though traditional databases satisfy most of the needs, they do not provide the sufficient speed performance.

- **Chronicle Map**[26] – provides a Java Map like interface and stores data off-heap. The data can be persistent by the use of memory mapped files[27]. On Linux machines it takes an advantage of lazy page allocation so the memory and disk usage can be small if it contains only a few entries. It is distributed under the GNU GPLv3 license [Free Software Foundation, 2007]. Disadvantages are that some hints are needed for the data structure to perform efficiently:
  - Number of entries – can be overestimated. However, the authors specify that it would be best to specify this number to the actual number of entries stored $\pm 10\%$. Even though it is possible to store more elements than specified, the performance of the data structure decreases marginally.
  - Average key size – does not affect maximum key size.

  Another disadvantage is that some Java options are needed for it to work with Java 9 and later releases.

- **MapDB**[28] – similar to Chronicle Map but no hints are necessary. It is distributed under the Apache license [Apache Software Foundation, 2004].

- **LMDB for Java**[29] – Java mapping for LMDB[30]. Major drawbacks:
  - Low level API – keys and values need to be passed in byte buffers.
  - Hints needed:

[25] `https://redis.io`
[26] `https://chronicle.software/products/map/`
[27] `https://en.wikipedia.org/wiki/Memory-mapped_file`
[28] `http://www.mapdb.org`
[29] `https://github.com/lmdbjava/lmdbjava`
[30] `https://symas.com/lmdb/`

    - Number of entries – can be overestimated.
    - Maximum key size – must be the same as the maximum term size in Lucene which is 32767 bytes.
  - Some Java options needed for Java 9 and later.

  It is distributed under the Apache license [Apache Software Foundation, 2004].

Benchmarks comparing MapDB, Chronicle Map, LMDB for Java and some other solutions performed by the LMDB for Java creators can be found in [Alex].

A benchmark for similar usage as in our case comparing MapDB, Chronicle Map and LMDB for Java can be seen in the Figure 2.11. The performed test consists of getting the integer value for every 20th term of *English words* dataset. The details on how the benchmark was performed and how it can be replicated can be found in Attachment A.

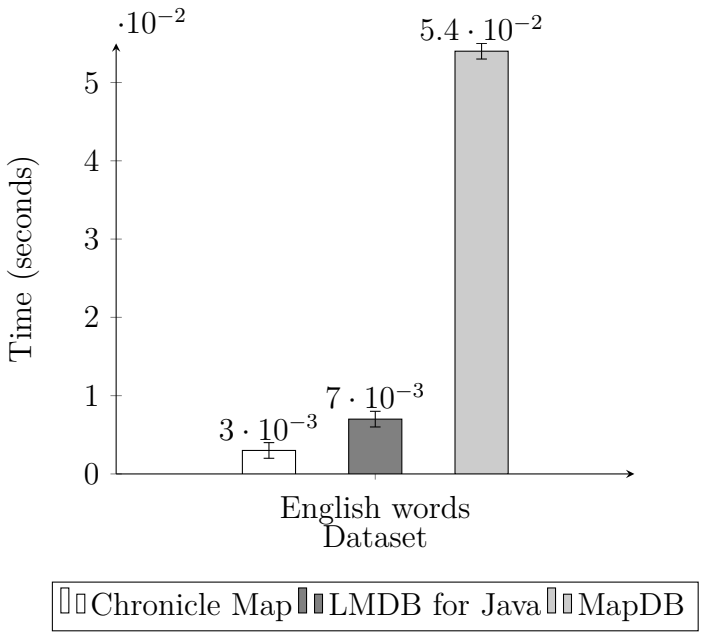

Figure 2.11: Comparison of lookup times for selected key-value stores

Although MapDB provides the nicest high-level interface and there is no need to specify its size in advance, its lookup speeds were lacking behind the other options. In the end, Chronicle Map was chosen to serve as a key-value store because it had proven to have a very solid performance.

As mentioned in the Section 2.3.1, the best option for simple prefix queries is a WFST data structure. Mainly because of its low memory footprint and fast lookups. Every term has a score assigned based on its occurrence frequency as mentioned in the Section 2.3.1. This score can be computed at the start and does not change very often – only after performing reindex. However, this changes with the use of the most popular completion. The data needs to be updated after every search. The best and easiest solution would be to change the term score in WFST directly. However, this is not possible because Lucene's implementation of WFST is read-only. Let's examine possible modifications to WFST to allow changing the score of the terms.

- WFST is stored in a byte array (*byte[]*) or multiple byte arrays to allow low memory footprint. The arc values are very often small because they only represent difference to the smallest term in the subtree. Lucene implementation leverages this and for small values less bytes need to be written. For instance, take an *int* data type – it needs 4 bytes to be stored properly. However, in the WFST implementation it takes between 1 and 5 bytes. The smaller values need less space. Per byte, the highest order bit indicates whether the next byte is part of this number and the remaining 7 bits contain the actual value. Even though this allows for a very efficient memory requirements, the modifications are a problem. If we were to increase some value, then it might take more bytes than it actually has available. Therefore, part of the byte array would need to be shifted by 1 byte. This operation would prevent parallel access. One solution might be that there would be another array where the content would be copied and then the references would be swapped. However, this does not allow parallel modifications. Another solution is to discard the variable length of the values and allocate a constant size for them, e.g. 4 bytes for *int*. This however increases the memory footprint as can be seen in the Figure 2.12.

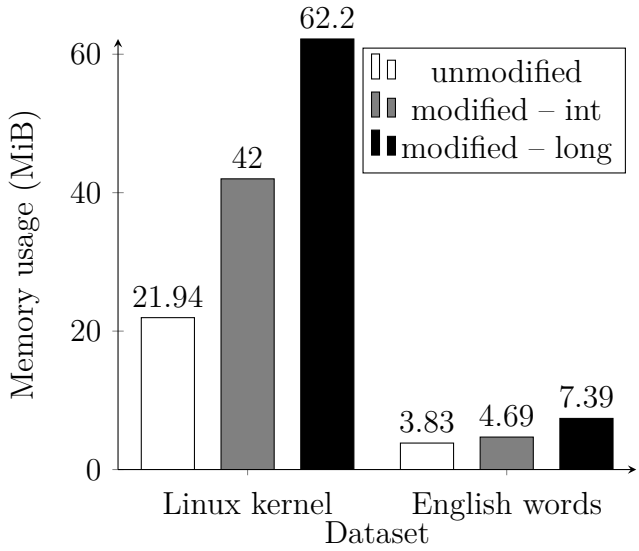

Figure 2.12: Comparison of memory usage when changing number encoding

The memory usage increased by approximately 22 % for *English words* dataset. However, it can be almost doubled as can be seen on *Linux kernel* dataset where approximately 92 % size increase can be noted. The graph in the Figure 2.12 also shows the case when the encoding would use *long* datatype. Although Lucene's *Lookup* interface specifies *long* datatype, WFST implementation supports only *int* so far.

- Lucene's WFST implementation does not know the notion of nodes – the data are stored only in arcs. The arcs that start in the root node might be stored in memory directly and therefore are not encoded in a byte array since these arcs are accessed very often and the decoding from a byte array has an impact on the data structure lookup performance. Although this is

not a big issue, the modifications would need to take this into account and add checks for this case.

- In some cases (e.g. the depth of an arc is less than a constant or there is more arcs from one node than another constant) the arcs might not be encoded as efficiently as possible but rather take a constant size. This then allows to do a binary search of arcs rather than going over them sequentially. Thus, increasing the speed of the lookup.

- As explained in the Section 2.3.1, scores of the terms are inverted (Equation 2.3) so by increasing score of some term we are actually decreasing its weight in WFST. Sum of arc values on the path to a term is the weight of the term (plus the final output modifier). However, the arc value tries to represent the lowest weight of the term in its subtree. And by decreasing this weight it could be necessary to update the whole WFST data structure. For instance, let's suppose that we have WFST as in the Figure 2.3 and that the weight of the term CAR will change to value 2. The changes can be seen in the Figure 2.13 in red color.

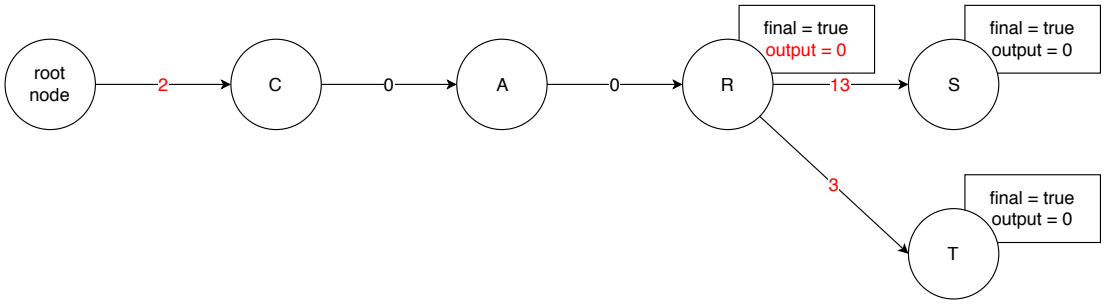

Figure 2.13: WFST modifications

As a consequence of these observations, it was decided that modifying the WFST to allow changing scores could be a difficult endeavour. Therefore, the remaining option was to rebuild WFSTs per some time interval. The most negative impact is that the suggestions for simple queries will not take the other searches into account immediately but only after some non-trivial time period.

This time period should be configurable. There are a few options from which to choose:

- **Period** – specified in some time unit. For instance, data should be rebuilt every hour. This solution is easy to implement. The period could start ticking after the last rebuild. The main drawback is that there is no way to specify the exact time.

- **Exact time** – is a good option when it is known when the system load will be small (e.g. during the night).

- **Cron format**[31] – cron is a job scheduler for Unix operating systems. When and which jobs should be run is specified in the special file called *cron table*. The problem is that the Java does not have native support for processing this format and some library must be used, the most notable are:

[31] `https://en.wikipedia.org/wiki/Cron`

- cron4j[32] – a simple library which allows to create a simple scheduler with a specified cron format and *Runnable*[33] that is to be performed. The provided interface is clean and to the point. However, at the time of writing, the library is no longer maintained and has not been updated in 5 years.
- cron-utils[34] - a small library for parsing and converting various cron formats. It also provides a feature which allows to compute the time to the next execution based on the cron format.
- quartz[35] – is an enterprise job scheduler library which allows to create truly complex jobs.

Chosen solution – it was desired to allow the administrators the most freedom. For that, the cron format was deemed the best option. It was a hard choice to pick between the mentioned libraries. In the end, *cron4j* was ruled out because of its oldness and the *quartz* because of its complexity. *cron-utils* can be quite easily combined with *ScheduledExecutorService*[36] to create the needed functionality.

Most Popular Completion – Complex Queries

There are multiple possible approaches how to enhance the suggester with the use of complex queries:

- Different term boosting – boost terms based on their popularity but also take into account other query parts. For example, if the search would be restricted to Java file types, then the suggester could only boost terms which were searched for when this condition held true.
- Suggest the whole complex query – if the user starts to write a part of a complex query then it could be recognized by the suggester which could suggest this already known query. However, this might not be user friendly since there are multiple search fields, content of which would have to be modified. Or the whole query would need to be written by the use of field specifiers which might make the query badly readable.

Possible implementation of both suggested approaches would require additional data structures and could slow down the suggester. For now, they are deemed only as a possible extension.

As a result, the complex query is only dismantled into simple term queries which are used to update the term popularity counts.

Increasing Search Counts for Specific Terms

Administrators might already have some search data stored. These data could be used to initialize or update the popularity term counts.

[32] `http://www.sauronsoftware.it/projects/cron4j/`
[33] `https://docs.oracle.com/javase/10/docs/api/java/lang/Runnable.html`
[34] `https://github.com/jmrozanec/cron-utils`
[35] `http://www.quartz-scheduler.org`
[36] `https://docs.oracle.com/javase/10/docs/api/java/util/concurrent/ScheduledExecutorService.html`

There are 2 possible approaches how to do this:

- By processing HTTP search request URLs:
  - Target endpoint:
    - Request method: POST
    - Endpoint: /api/v1/suggest/init/queries
    - Media type: application/json
  - Example data:

```
["http://demo.opengrok.org/search?project=opengrok&q=test"]
```

- By processing JSON data:
  - Target endpoint:
    - Request method: POST
    - Endpoint: /api/v1/suggest/init/raw
    - Media type: application/json
  - Example data:

```
[{"project": "opengrok", "field": "full", "token": "test",
"increment": 1}]
```

User Specific Suggestions

The Suggester might be further extended to provide suggestions based on the user specific previous searches. Google does something similar as is highlighted by the number 1 in the Figure 2.14.

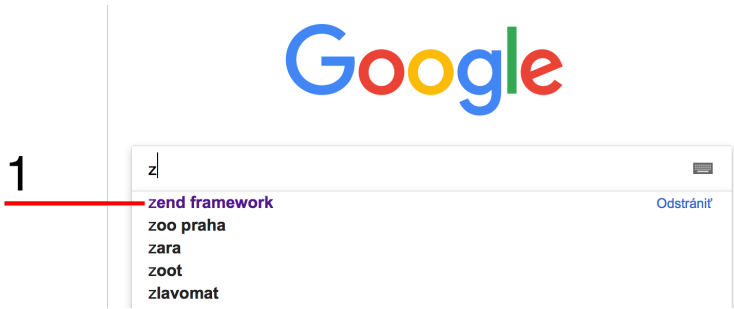

Figure 2.14: Suggestions based on the user specific previous searches by Google

Since authorization is optional in OpenGrok, it would be best to store the previous search data in browser cookies[37]. This data could then be processed and prepended to the returned suggestions from the Suggester. Furthermore, it might be possible to combine personalized suggestions with the relative query popularity as described in [Cai et al.]. However, the implementation of this feature is considered as a possible extension.

[37] https://en.wikipedia.org/wiki/HTTP_cookie

2.3.11 Promoting Suggestions Based on the Code Properties

OpenGrok is a search engine for codebases which could be leveraged by the suggester. Following are things that could be taken into account:

- **Indentation** – the more the line is indented, the less significant it probably is.
- **Type** – the term might have different importance based on its type in the code. For example, a class name, method name, global variable name, or local variable name could have different importance factors.

However, this might be difficult to implement and is presented only as a possible extension for the future.

2.3.12 Do not Suggest the Same Term Already in the Query

It can easily happen that user selects the first suggested item for some prefix p. If the user were to write the same prefix again then the suggester should not offer the term that is already in the query.

For instance, imagine that the search field contains value $c|$ ($|$ signifies the input caret). The suggester suggests values *cat, color*. If the user chooses *cat* and again types c then the result query has format *cat* $c|$. In this case the suggester should not suggest the *cat* value but only the *color* value.

2.3.13 Authorization

As specified in the Section 1.1, OpenGrok allows the administrators to specify to which projects the user has access. Therefore, suggester must not allow unauthorized users to see suggestions because it would be considered a security issue.

2.3.14 Restricting Load

The Suggester may consume a significant amount of resources for some specific queries (mainly Phrase Queries). Therefore, the Suggester should respect the paradigm "best effort"[38]. The suggestions are not an integral part of the search engine. Thus, if their resource requirements are too high, then they should not prevent users from using the basic OpenGrok functionality. As a result, the Suggester will try to find suggestions for some query up to some pre-defined time threshold and if it is not able to complete the request in this time limit, then it will return only partial result. This also solves the problem of an attacker who would send many requests containing the worst queries for some dataset.

[38] `https://en.wikipedia.org/wiki/Best-effort_delivery`

2.3.15 Configuration

As mentioned in the Section 1.2.2, OpenGrok can be configured using a configuration file. The Suggester should be also configurable to allow the administrators to specify how they want the suggester to perform. Instead of creating its own separate way to handle configuration, it will depend on the already existing solution in OpenGrok. The main advantage is that the administrators are already familiar with how the configuration works. Configurable properties of the suggester:

- **Enabled** – specifies if the suggester is enabled. Some users might not want the suggester functionality. Default value: *true*.
- **Maximum results** – specifies how many results the suggester should return at maximum. Default value: *10*.
- **Minimum characters** – specifies the minimum number of characters that are needed for the suggester to start looking for suggestions. More characters mean less possible candidates. It can significantly improve performance of the suggester. Default value: *0*.
- **Allowed projects** – specifies a set of projects for which the suggester should be enabled. Default value: *all*.
- **Maximum projects** – specifies how many projects can be selected at the same time and the suggestions will work. Default value: *unlimited*.
- **Allowed fields** – specifies the fields for which the suggester should be enabled. For instance, it might be desired that the suggestions should only work for the *full* field. OpenGrok uses some fields to store private data and it is not desired to create suggester data structures for those fields. Therefore, only the main search fields are specified by default. Default value: *full, defs, refs, path, hist, type*.
- **Allow complex queries** – specifies if the suggester should support complex queries. If set to *false* then only simple prefix lookups by using the WFST data structure will be performed. Default value: *true*.
- **Allow most popular** – specifies if the most popular completion should be enabled. If set to *false* then it slightly increases the performance and there is no need for WFST rebuilds. Default value: *true*.
- **Show scores** – specifies if the scores should be displayed next to the suggestions. Default value: *false*.
- **Show projects** – specifies if the suggestions should show in which project the term was found. If there are multiple projects then showing all the names is not feasible; therefore, only the number of projects will be shown. Default value: *true*.
- **Show time** – specifies if the time it took the suggester to find the suggestions should be displayed. Default value: *false*.

- **Rebuild cron config** – specifies how often should the suggester rebuild the WFST data structures. The value must be in the Unix cron format, decision is described in the Section 2.3.10. Default value: *0 0 * * *** (every day at midnight).

- **Build Termination Time in Seconds** – specifies after how much time the suggester should kill the threads that build the suggester data structures. Slow machines should specify more time. This option is mainly here to prevent the suggester to hang in the initialization. Default value: *1800* (30 minutes).

- **Time threshold** – specifies a time threshold for suggestions in milliseconds. If the computation exceeds this time, it will be stopped and partial results will be returned. Default value: *2000* (2 seconds).

2.3.16 Tolerating Errors

Although it is possible to use fuzzy factor as specified in the Section 2.3.8, the Suggester is not error tolerant. This could be a good extension to the prefix queries (Section 2.3.1). For instance, if the user types the prefix *sh* even though the index contains the term *schwarz*, it won't be among the suggestions. This might prove very useful if the user made a typing error or is not sure about the spelling of the term.

[Chaudhuri and Kaushik, 2009] studies the possibility of tolerating errors for prefix autocompletion. Authors defined prefix p as a *k-prefix* of string s if there is some extension of p that is within edit distance k of s. s is called a *k-extension*. Then, for some prefix it is possible to consider its k-extensions up to some constant k. Authors presented Trie and Q-gram based algorithms. However, this would need another data structures and does not work for other types of queries. Therefore, it is considered as a possible extension.

2.3.17 Retrieving Popularity Data

In the Section 2.3.10 it was discussed that most popular completion will be used. Many users or administrators might be interested in the terms popularity. Therefore, administrators can access this information via the REST API by accessing */api/v1/suggest/popularity/{project}* endpoint. The *{project}* represents a project name for which to retrieve the data. Additional supported query parameters:

- **field** – field for which to retrieve data. Default: *full*.

- **page** – page of data to retrieve.

- **pageSize** – how many results will be included in the page.

- **all** – if specified to `true`, then all the data are retrieved.

The returned data are in the *JSON* format and are sorted by the popularity in descending order. Example of returned data:

```
[{"semaphore":2},{"mmap":1},{"test":1}]
```


As mentioned, this functionality is only enabled for administrators. If the need arises, it should be possible to add functionality which presents this data to users with minimum effort.

3. User Documentation

This chapter describes how the suggester should be used. It is divided into 2 sections:

- 3.1 User guide – usage described from the end user point of view.
- 3.2 Administrator guide – describes how to modify the suggester configuration.

3.1 User Guide

When the OpenGrok main web page is opened the user sees something similar to the Figure 1.2. When the user types text into one of the search fields, suggestions are shown as can be seen in Figure 3.1.

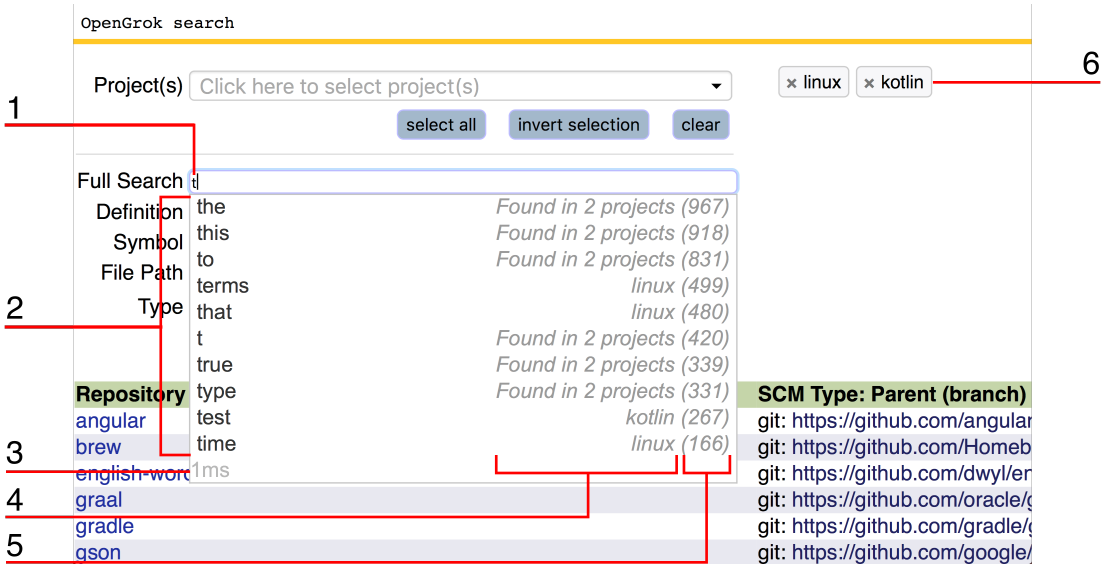

Figure 3.1: Suggestions example

The numbers in the Figure 3.1 represent:

1. Typed value into the search field. In this case letter *t*.
2. List of suggestions.
3. Time it took to generate the suggestions. (Optional.)
4. In which projects the term was found. (Optional.)
5. Score of the terms. (Optional.)
6. Projects for which suggestions are shown.

3.1.1 Selecting the Suggestions

The suggestions can be navigated by using:

- **Keyboard** – the suggestion can be focused by using up ↑ and down ↓ arrow keys. The suggestion can be selected by pressing the *enter* (carriage return) or *tab* key. The suggestions window can also be closed by pressing the *esc* key.
- **Mouse** – by hovering over the suggestion it is focused. By clicking on the suggestion it is selected.

Focused Suggestion

If a suggestion is focused, then its background color is changed to *blue*. In case of keyboard focus, the text of the field shows the result value as if this suggestion was selected. It looks like in the Figure 3.2.

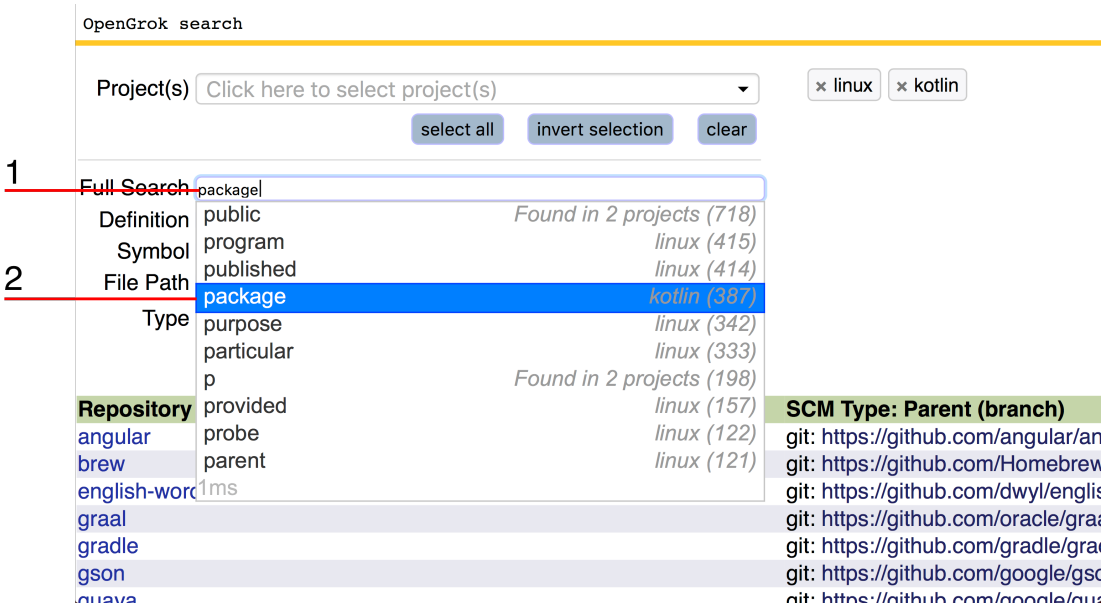

Figure 3.2: Focused suggestion

The numbers in the Figure 3.2 represent:

1. The field already contains the value *package* even though only the prefix *p* is actually written.
2. Focused suggestion.

Errors

It might be a common occurrence that users type an invalid query. For instance, *)* is missing at the end of the query *(test*. Therefore, the user is notified if the Lucene cannot parse the query or some other error occurs. This case is pictured in the Figure 3.3.

The numbers in the Figure 3.3 represent:

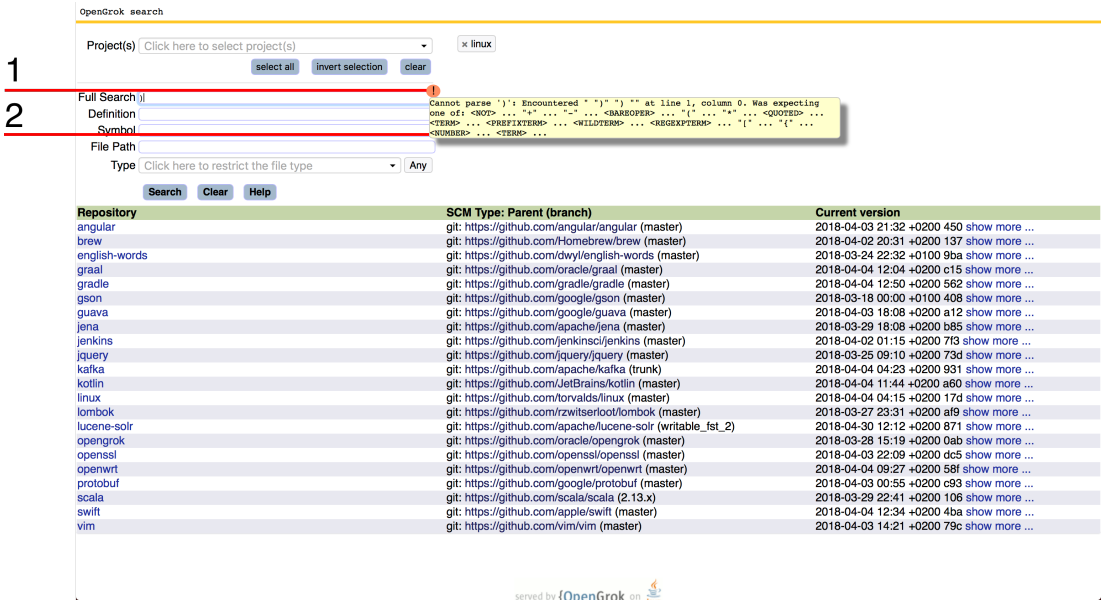

Figure 3.3: Suggestions error

1. Error notification.
2. Detail of the error. This is only shown if the user hovers over the error notification.

Minisearch

While traversing a source code or history, a minisearch field is available in the navigation menu. It allows users to search in the current project. The suggestions are enabled for it as well. The result can look like in the Figure 3.4.

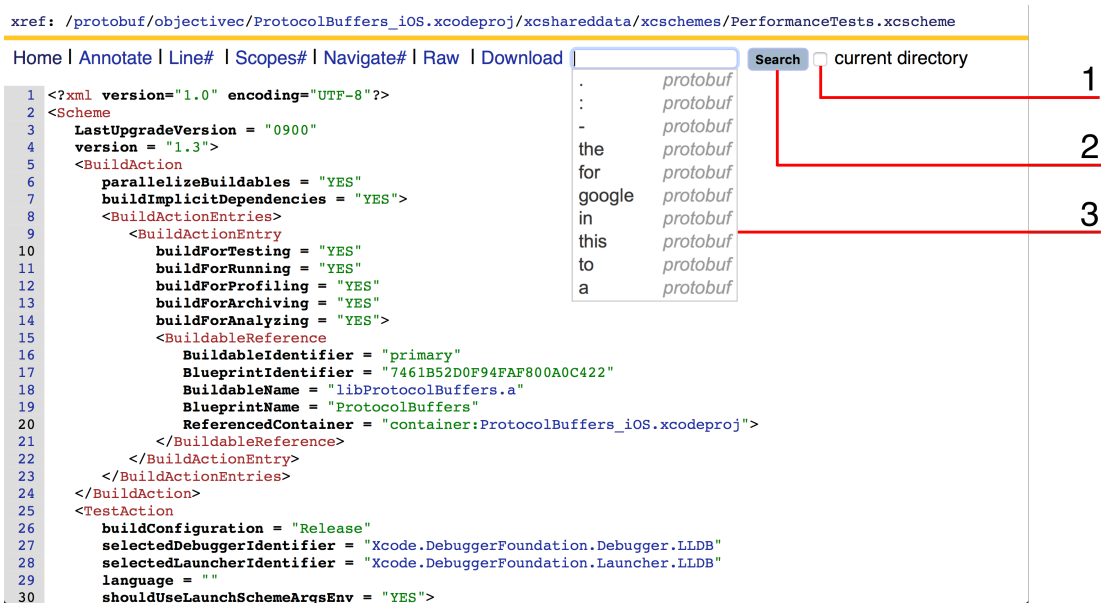

Figure 3.4: Suggestions for minisearch

In the Figure 3.4 the numbers represent:

1. Checkbox that specifies if the search (and consequently suggestions) should be restricted to the directory in which the displayed file is located. In this example it adds the following value to the *path* field:

```
"/protobuf/objectivec/ProtocolBuffers_iOS.xcodeproj/xcshareddata/
xcschemes/"
```

2. Search button that launches the search.

3. Suggestions window as described before.

Timeout

As specified in the Section 2.3.14, the system may be under load and it might return only a partial result or no suggestions at all. In the case of the partial result, the message *"Partial result due to timeout"* will be displayed at the bottom of the suggestions window similarly as in the Figure 3.5.

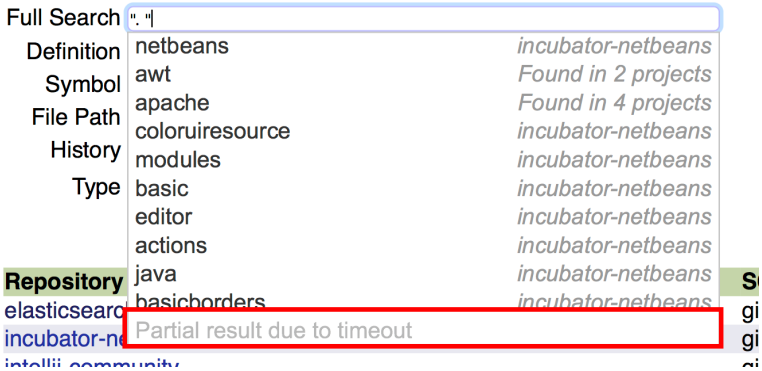

Figure 3.5: Message if only partial result was returned

If no suggestions were returned, then it may signify two cases:

- The query yields no results. In that case a message *"No matches found"* will be displayed as in the Figure 3.6.

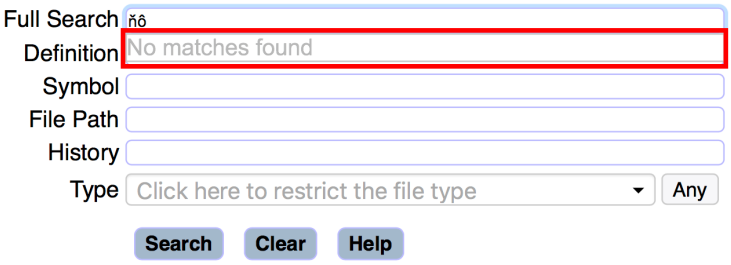

Figure 3.6: Message if no matches were found for the specified query

- The system timed out and did not find any possible suggestions. In that case no suggestions window will be displayed.

Support for Lucene's Field Modifier

Lucene queries can contain a field modifier which specifies the field for which the query should be matched (syntax {*field*}:{*query*}). OpenGrok actually performs this modification when the user types queries into different search inputs. However, it is also possible to use these modifiers in the *Full Search* input field. For example, it is possible to write a query `defs:a|` (| represents a caret position) and the suggester will suggest terms with prefix `a` exactly as it would if the text `a|` were written into the *Definition* input.

3.2 Administrator Guide

There are multiple ways to modify the suggester configuration described in the Section 2.3.15:

- By modifying the OpenGrok's XML configuration described in the Section 1.2.2. An example:

```
<java version="1.8.0_162" class="java.beans.XMLDecoder">
  <object
      class="org.opensolaris.opengrok.configuration.Configuration"
      id="Configuration0">
    <void property="suggesterConfig">
      <void property="allowComplexQueries">
        <boolean>false</boolean>
      </void>
      <void property="allowMostPopular">
        <boolean>false</boolean>
      </void>
      <void property="allowedFields">
        <object class="java.util.Collections" method="singleton">
          <string>defs</string>
        </object>
      </void>
      <void property="allowedProjects">
        <object class="java.util.Collections" method="singleton">
          <string>fruitonserver</string>
        </object>
      </void>
      <void property="enabled">
        <boolean>false</boolean>
      </void>
      <void property="maxProjects">
        <int>2</int>
      </void>
      <void property="maxResults">
        <int>11</int>
      </void>
      <void property="minChars">
        <int>2</int>
      </void>
```

```
      <void property="rebuildCronConfig">
        <string>5 * * * *</string>
      </void>
      <void property="showProjects">
        <boolean>false</boolean>
      </void>
      <void property="showScores">
        <boolean>true</boolean>
      </void>
      <void property="showTime">
        <boolean>true</boolean>
      </void>
      <void property="buildTerminationTime">
        <int>500</int>
      </void>
    </void>
  </object>
</java>
```

- By using the REST API configuration endpoint. It is possible to send a JSON representation of the suggester configuration. Only the properties which differ from the default configuration need to be specified. Example data:

  ```
  {"enabled": "false"}
  ```

  Suggester configuration endpoint:

  ```
  /api/v1/configuration/suggesterConfig
  ```

4. Program Documentation

This chapter focuses on the implementation part of the thesis.

4.1 Overview

As mentioned, suggestions are retrieved by a REST API call made to the Web module. This module processes the data and invokes the Suggester module to return the suggestions. Simplified diagram which shows the main interactions between the objects and modules can be seen in the Figure 4.1.

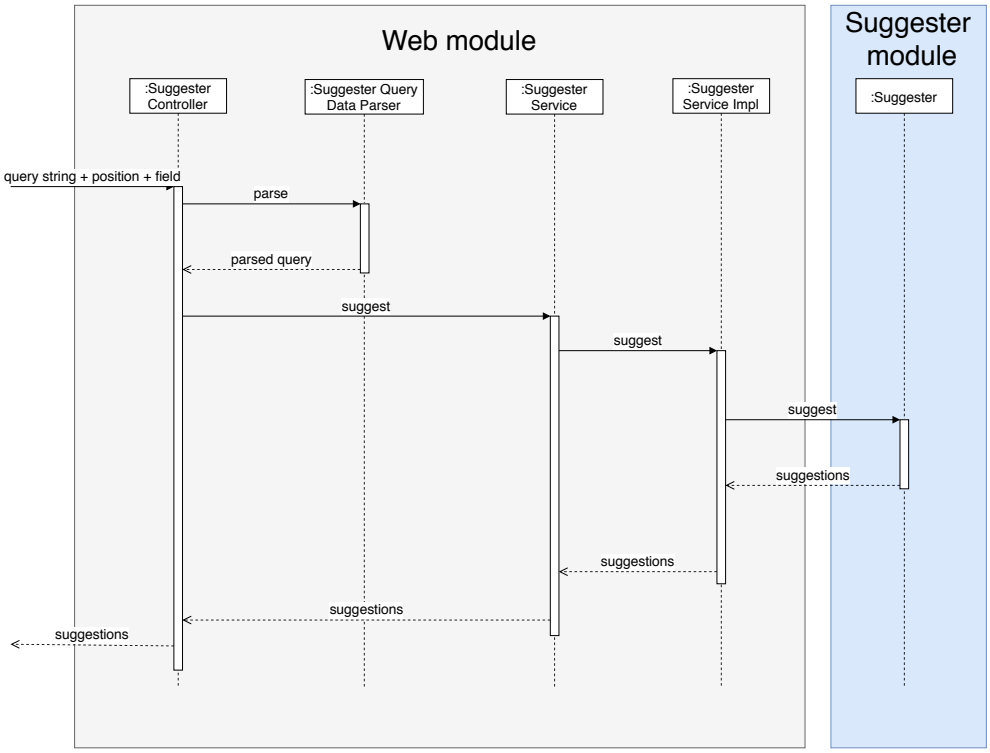

Figure 4.1: Overview of the main object interactions

4.2 Web module

The suggester support was added as a part to the already existing OpenGrok's REST API. The *SuggesterController* class serves as a REST API endpoint for suggester related queries. Other needed classes are located in the *org.opensolaris.opengrok.web.api.v1.suggester* package. Classes which are auto-detected by the Jersey implementation are located in the *org.opensolaris.opengrok.web.api.v1.suggester.provider* package and are annotated by the *@Provider*[1] annotation. The communication with the Suggester module is abstracted in the *SuggesterService* interface and implemented in the *SuggesterServiceImpl* class. The implementation is injected into the classes which require the *SuggesterService* by the use of

[1] `https://docs.oracle.com/javaee/7/api/javax/ws/rs/ext/Provider.html`

@Inject[2] annotation. If some other component which is not part of the REST API part of the application needs to access the *SuggesterService*, it can retrieve it by invoking *getDefault()* method in the *SuggesterServiceFactory* class.

Configuration Change

The suggester data structures need to be refreshed on configuration change. Many basic OpenGrok properties might have changed, for instance:

- *dataRoot* which specifies where OpenGrok stores the data.
- Suggester configuration.
- Projects.

It is not trivial to notice all the changes and it would not be feasible from the modifiability point of view to check specific values for change. Therefore, *SuggesterServiceImpl* closes the underlying suggester data structures and tries to reinitialize the suggester with the new configuration. If the configuration has not changed in a major aspect (e.g. different *dataRoot* or new projects were added), then the Suggester module just reloads the data from disk. Reload from disk should be fast (from the experience less than 1 s).

4.3 Suggester module

The Suggester module is located under the *suggester* directory of the root of the project. The main class which exposes the suggester functionality is the *Suggester* class.

Public API

The public API consists of the following methods:

- *init(Collection<NamedIndexDir>)* – initializes all the data structures based on the paths to the indexes and their names (project names).
- *remove(Iterable<String>)* – removes all the data structures stored for the specified names.
- *search(List<NamedIndexReader>, SuggesterQuery, Query)* – searches the suggester data and returns suggestions to the provided queries.
- *onSearch(Iterable<String>, Query)* – event signalization that the query was searched. Updates the data structures for the most popular completion.
- *increaseSearchCount(String, Term, int)* – updates the search count for the specified project and term by the specified int value. Negative values are not allowed.
- *getSearchCounts(String, String, int, int)* – returns popularity data for the specified project and field.
- *close()* – closes the suggester data structures and other functionality.

[2] `https://docs.oracle.com/javaee/7/api/javax/inject/Inject.html`

Suggester Data

The following shows the typical *dataRoot* content for a multi-project OpenGrok setup along with the suggester data.

```
dataRoot ............... OpenGrok's data root specified in the configuration
├── historycache ..................................... history data for files
│   ├── project1 ........................... history data for files of project1
│   │   └── ...
│   └── ...
├── index ............................................... index data
│   ├── project1 ................................... index data for project1
│   │   └── ...
│   └── ...
├── statistics.json ............................... stored statistics data
├── suggester ............................................ suggester data
│   ├── project1 ............................... suggester data for project1
│   │   ├── {field}_map.cfg ............ stored Chronicle Map configuration
│   │   ├── {field}_search_count.db ................ stored Chronicle Map
│   │   ├── {field}.wfst ...................... stored WFST data structure
│   │   └── version.txt ....................... version of the suggester data
│   └── ...
├── timestamp ................................ timestamp of last indexing
└── xref ...................................... pre-generated HTML files
    ├── project1 ................... pre-generated HTML files for project1
    │   └── ...
    └── ...
```

In the case of a single-project OpenGrok setup, there is no *project1* as specified in the previous example but the data are stored directly in directories *historycache, index, suggester* and *xref*.

The {*field*} represents the Lucene field for which the files contain the data, e.g. *full*. If some data are corrupted and the suggester is not able to read them, the best solution might be:

1. If possible, then backup the popularity data as specified in the Section 2.3.17. (Not needed if popularity data is not important or turned off.)

2. Stop the web application.

3. Remove either the corrupted data or the whole suggester directory.

4. Start the web application again.

5. Initialize the popularity data as specified in the Section 2.3.10. Some data modifications would be needed; however, if the need arises, an automatic tool might be created for this purpose. (Not needed if popularity data are not important or turned off.)

Chronicle Map Configuration

The Chronicle Map does not remember the configuration (number of entries, average key size) with which it was created. Therefore, this information needs to

be stored. Exactly for this purpose serves the aforementioned {*field*}_*map.cfg* file. For storing and loading this information, the *ChronicleMapConfiguration* class was created. It is a *Serializable*[3] class which contains the needed information.

Detecting Index Version

The Indexer might run even if the Web application is turned off. Therefore, it would not be possible to notify the Suggester about the change. As a result, the Suggester needs to detect this by itself. Exactly for this purpose serves the aforementioned file *version.txt*. It contains a number which specifies the generation of the last index commit for which the data were created. Upon detecting that the data version does not match with the index version, the suggester will rebuild its data by itself.

Data Structures Abstractions

The implementation contains interfaces for data structures so the implementation might change and there would be no need to modify other parts of the Suggester module. Abstract data structures:

- **PopularityMap** - abstraction for most popular completion.
- **IntsHolder** – abstraction for holding a set of positions in a document used for Phrase Query evaluation.

Use as a Separate Library

Because of the decision that the Suggester module is not dependent on any of the OpenGrok modules, the Suggester module could be used in different projects that use Lucene as a simple library. However, a few modifications might be needed to make it more user-friendly, namely:

- Remove the *projectsEnabled* parameter from the constructor. It is OpenGrok specific.
- Overload the method *search(List<NamedIndexReader>, SuggesterQuery, Query)* to provide the possibility to search without the need for the list of the *IndexReader* variables. They are provided now to better facilitate the resource reuse. However, they are not needed and could be created from the index paths specified in the *init(Collection<NamedIndexDir>)* method. The new method could have the signature *search(List<String>, SuggesterQuery, Query)* which would require only index names.
- Provide a default parser which would be able to create *SuggesterQuery* instances. This could be a modified version of *SuggesterQueryParser* already present in the Web module.

[3] `https://docs.oracle.com/javase/10/docs/api/java/io/Serializable.html`

4.4 Testing

Unit tests[4] were written for the Suggester module to test the basic functionality. They can be found in a standard Maven test directory *src/test/java*.

To test the real word usage and to test the integration with other OpenGrok modules, tests were written for *SuggesterController* which test the suggester endpoint in the Web module which in turn calls various parts from Indexer and Suggester modules. Small projects written in variety of programming languages used for OpenGrok's own testing were used as the test data. These can be found in the *testdata/sources* directory of the OpenGrok project root. First of all, these projects are copied into the temporary *sourceRoot* directory. Then they are indexed. After that, the Suggester is initialized. Finally, the tests run.

However, the tests are not exhaustive and for better test coverage more tests should be written.

Regarding UI testing, OpenGrok does not have any UI tests in place. Therefore, it was not deemed to be a high priority. In the future, it would make sense to incorporate at least some UI testing.

[4] `https://en.wikipedia.org/wiki/Unit_testing`

5. Impact on the OpenGrok Project

Change impact analysis[1] is a field of study, the task of which is to determine what impact the changes may have on the system. Sometimes, it is very difficult to analyze the impact of a change. Even more so in large projects with huge codebases. Analysis of the main decisions made is described in detail in the Chapter 2. The main aspects of the added changes that can be discussed are:

- Impact on the search results. Discussed in more detail in the Section 5.1.
- Impact on the hardware requirements. Discussed in more detail in the Section 5.2.

5.1 Impact on the Search Results

It is hard to compare search results and search engines in general. Which search engine performs better: the one that returns only a few relevant documents but omitted many in the process or the one that returns all relevant documents but includes some that are not? In the information retrieval field there are measurements which take this into account, the most notable are **precision** and **recall**[2].

Precision is the fraction of retrieved documents that are relevant to the query. See Equation 5.1.

$$precision = \frac{|\{relevant\ documents\} \cap \{retrieved\ documents\}|}{|\{retrieved\ documents\}|} \tag{5.1}$$

Recall is the fraction of the relevant documents that are successfully retrieved. See Equation 5.2.

$$recall = \frac{|\{relevant\ documents\} \cap \{retrieved\ documents\}|}{|\{relevant\ documents\}|} \tag{5.2}$$

However, the suggester does not impact these measurements directly. The same query returns the same results with or without the suggester. Nevertheless, the suggester affects them indirectly:

- There are less queries which yield no results – if the user types a few characters and there are no suggestions then he can immediately see that there are no terms with that prefix. Therefore, the result will be empty.
- Probably a negative impact on precision because of the scoring described in the Section 2.3.1. The terms which are in more documents have greater scores. Thus, these terms are promoted and are more likely to be chosen by the user. Therefore, the size of the retrieved document set will be greater. However, the initial scoring might be mitigated by taking into account previous users' searches as described in the Section 2.3.10.

[1] `https://en.wikipedia.org/wiki/Change_impact_analysis`
[2] `https://en.wikipedia.org/wiki/Precision_and_recall`

It is not easy to achieve feasible values for both precision and recall. In many information retrieval systems when one is being increased the other one decreases. This is most notable in Boolean Information Retrieval Model[3].

5.2 Impact on Hardware Requirements

The most affected resources are:

- **CPU** – in simple situations where only a prefix is typed, the CPU load is not that high because it is a lookup in the WFST data structure which is optimized for this kind of scenarios. However, in other cases index searches are performed which can consume a lot of CPU time.

- **Memory** – the WFST data structures are held in memory. Although their memory footprint is very low, one data structure needs to be created per Lucene field per project which can sum up to a significant value. Also, data for most popular completion are stored in the Chronicle Map implementation which translates to additional memory consumption.

- **Disk** – the WFST data structures are stored on the disk to provide a quick startup. The data for most popular completion need to be stored as well. The comparison of disk consumption for different datasets can be seen in the Figure 5.1. The data show how much percentage of the index size the suggester data take. The data were measured on the machine with the operating system macOS[4] and APFS[5] file system so the Chronicle Map did not take the advantage of lazy page allocation.

  Projects mentioned in the Figure 5.1 were:

  - **Apache Lucene and Solr** – `https://github.com/apache/lucene-solr`
  - **Apache Kafka** – `https://github.com/apache/kafka`
  - **GraalVM** – `https://github.com/oracle/graal`
  - **Jenkins** – `https://github.com/jenkinsci/jenkins`
  - **jQuery** – `https://github.com/jquery/jquery`
  - **Kotlin** – `https://github.com/JetBrains/kotlin`
  - **Linux kernel** – `https://github.com/torvalds/linux`
  - **OpenGrok** – `https://github.com/oracle/opengrok`
  - **OpenSSL** – `https://github.com/openssl/openssl`
  - **Swift** – `https://github.com/apple/swift`

[3] `https://en.wikipedia.org/wiki/Standard_Boolean_model`
[4] `https://en.wikipedia.org/wiki/MacOS`
[5] `https://en.wikipedia.org/wiki/Apple_File_System`

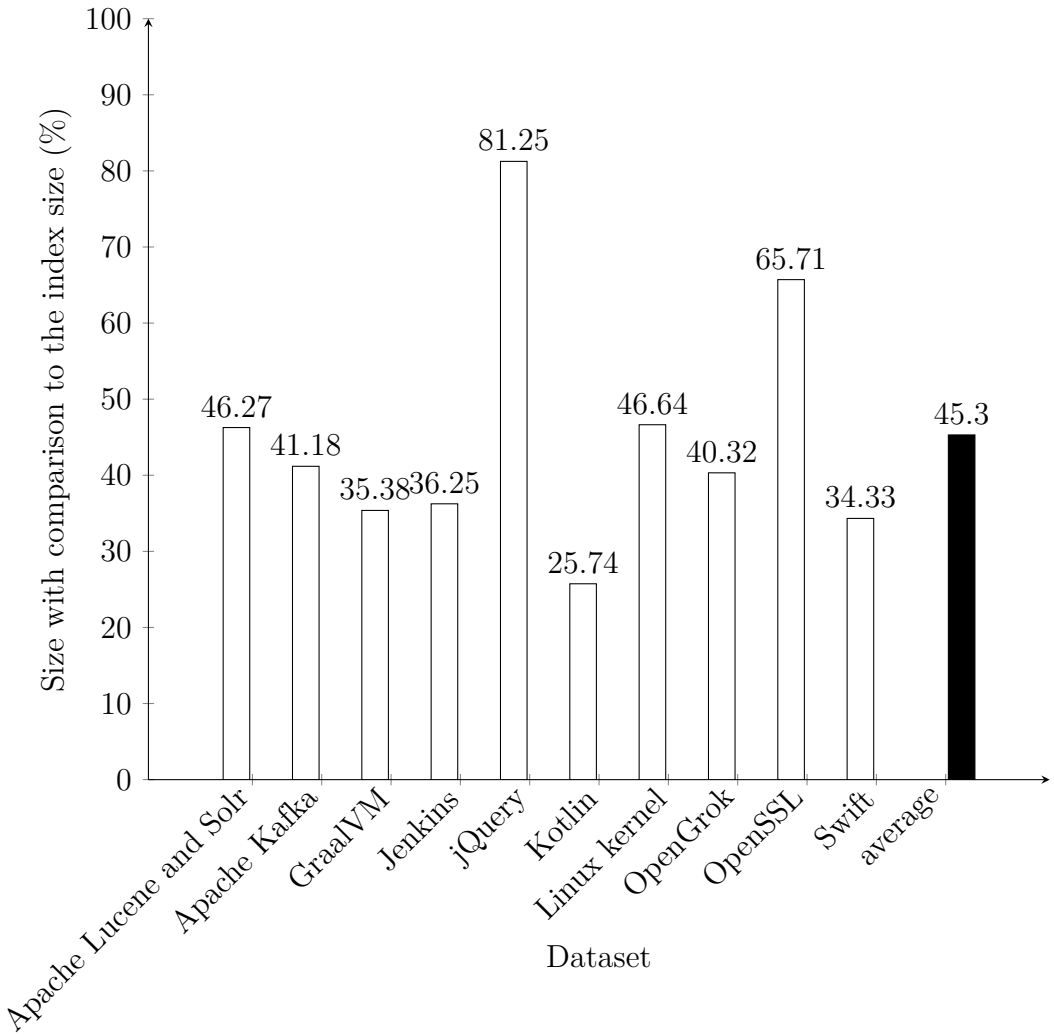

Figure 5.1: Comparison of suggester data size with index size

5.3 Impact on the Demo Instance

For monitoring the impact of the suggester, a demo instance has been created which can be accessed on the URL `http://demo.opengrok.org`. This instance contains 5 projects:

- **elasticsearch**[6] – scalable search engine built upon Apache Lucene.
- **NetBeans**[7] – NetBeans IDE in development under ASF[8].
- **IntelliJ IDEA Community Edition**[9] – open-source edition of the popular IDE.
- **Lucene/Solr**[10] – combined repository for Apache Lucene and Apache Solr.
- **OpenGrok**

However, real instances may, and often do, contain more projects.

The demo instance uses the Apache Tomcat servlet container. Many solutions exist which provide a possibility to monitor applications in a Tomcat instance. From those the most significant are:

- **JMX**[11] **remote** – Tomcat provides possibility to manage and monitor the applications via JMX. Many applications use this functionality to their advantage.
- **JavaMelody**[12] – can be added to the project as a dependency and creates a simple page with the monitoring information available at */monitoring* URL. Provides monitoring of CPU, memory usage, HTTP request count and many others. Also allows exporting data in various formats, e.g. PDF or JSON. The main disadvantage is that it is embedded into the application and may influence the gathered data. However, this overhead is small, e.g. the memory overhead did not exceed 3 MiB.
- **PSI Probe**[13] – deployed as a separate application into the Apache Tomcat instance. Uses JMX exported by Tomcat. The disadvantages are:
  - Cannot export the data in the common exchange formats. However, raw XML data are created under the Tomcat home directory.
  - Data are not persistent.

Chosen solution JavaMelody was chosen because of its simplicity and because it provided all the needed features despite the small overhead.

[6] `https://github.com/elastic/elasticsearch`
[7] `https://github.com/apache/incubator-netbeans`
[8] `https://en.wikipedia.org/wiki/Apache_Software_Foundation`
[9] `https://github.com/JetBrains/intellij-community`
[10] `https://github.com/apache/lucene-solr`
[11] `https://en.wikipedia.org/wiki/Java_Management_Extensions`
[12] `https://github.com/javamelody/javamelody`
[13] `https://psi-probe.github.io/psi-probe/`

5.3.1 Disk Usage

The demo instance is running on a machine with the Linux operating system. Therefore, Chronicle Map takes the advantage of lazy page allocation and the sizes are significantly smaller in comparison with the sizes in the Figure 5.1. The actual disk usage can be seen in the Figure 5.2. It should be noted that the Chronicle Maps were almost empty. Similar sizes might be expected as in the Figure 5.1 if the Chronicle Maps start to fill up.

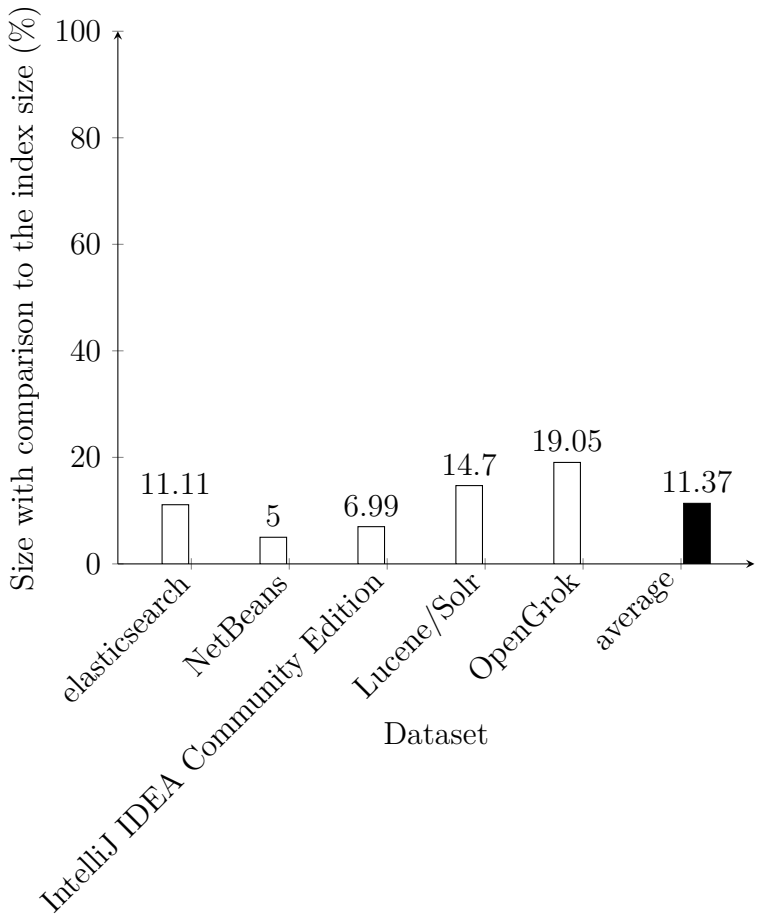

Figure 5.2: Comparison of suggester data size with index size on demo instance

5.3.2 Impact on CPU and Memory Usage

Impact on the CPU and memory usage along with the number of HTTP hits can be seen in the Figure 5.3. In the figure there are multiple colored parts which signify:

- The **red** area signifies the period during which the suggester functionality was enabled.
- The **purple** area signifies a peak that occurred because the timeout functionality had not been implemented yet and multiple users were writing the worst possible queries.

- The **green** histogram signifies the minimum values.
- The **blue** histogram signifies the maximum values.

It can be noted that the number of HTTP hits stayed roughly the same. There was not much traffic on the demo instance so the results might not be as accurate as would be desired. The memory usage did not change very much. However, it can be seen that the CPU usage maximum values increased.

5.4 Load Testing

Apache JMeter[14] was used for load testing. The testing was performed on the same machine as described in Attachment A. The JMeter's *.jmx* file used can be found in the project specified in Attachment A. 22 mostly well-known projects were used. 10 of them were the same as in the Section 5.2. The other twelve were:

- **Angular** – `https://github.com/angular/angular`
- **Apache Jena** – `https://github.com/apache/jena`
- **brew** – `https://github.com/Homebrew/brew`
- **english-words** – `https://github.com/dwyl/english-words`
- **google-gson** – `https://github.com/google/gson`
- **Gradle** – `https://github.com/gradle/gradle`
- **Guava** – `https://github.com/google/guava`
- **Lombok** – `https://github.com/rzwitserloot/lombok`
- **OpenWrt** – `https://github.com/openwrt/openwrt`
- **Protocol Buffers** – `https://github.com/google/protobuf`
- **Scala** – `https://github.com/scala/scala`
- **Vim** – `https://github.com/vim/vim`

The tests were run on the same machine as the running OpenGrok instance; therefore, the results might be impacted in a negative way. The properties of the used machine can be found in Attachment A.

[14] `https://jmeter.apache.org`

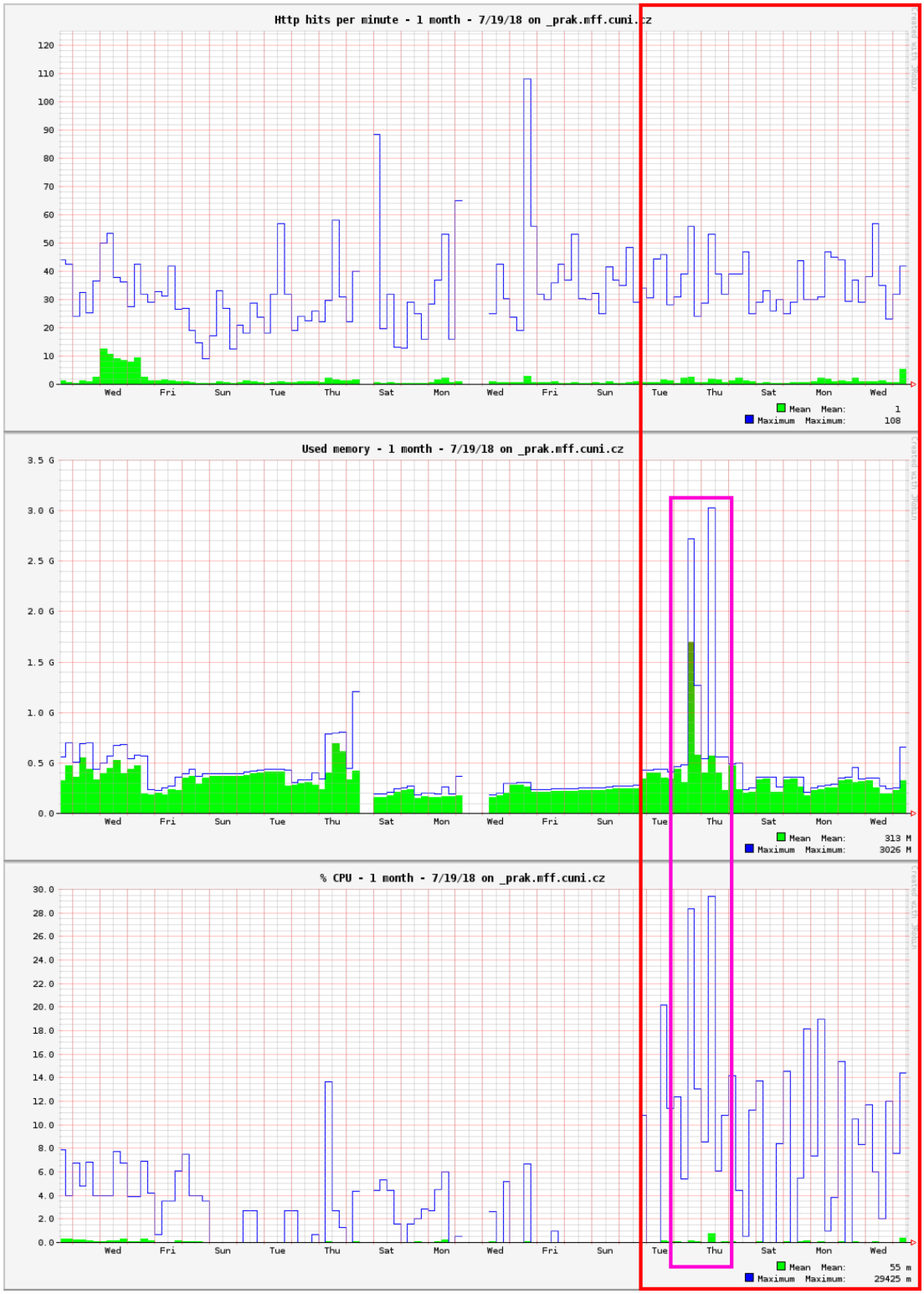

Figure 5.3: CPU usage, used memory, and HTTP hits metrics provided by JavaMelody during a month time period

5.4.1 Simple Prefix Query Across All Projects

This test case consists of 2,000 requests sent during a 10 second period. The requests were linearly distributed. To simulate a real-world use, the following queries were sent at random:

1. Prefix `long` in *full* field.
2. Prefix `m` in *path* field.
3. Prefix `D` in *refs* field.
4. Empty prefix in *defs* field.
5. Empty prefix in *full* field.
6. Prefix `c` in *full* field.

The requests specified all 22 projects. The results can be seen in the Figure 5.4. The slow startup can be noted; however, later requests were taking only a few milliseconds on average.

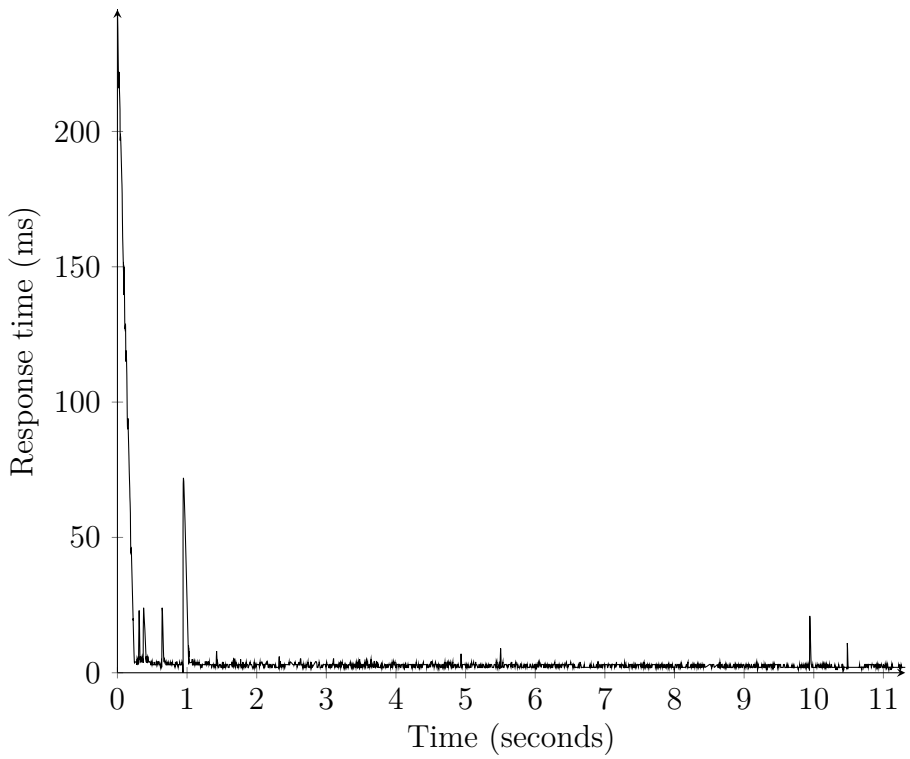

Figure 5.4: Load testing – prefix queries across multiple projects

5.4.2 Worst Case Across All Projects

Query `". |"` was used where | represents a caret position. Term . occurs in 205,943 files. All these files need to be checked for the . position and then all terms are traversed to check if they occur at the positions next to the . term. Time threshold was set to the default value: 2 seconds. The test case sends this

request 1,000 times linearly distributed during a 10 second time period. It can be noted that the system was not able to satisfy the requests; however, partial results were returned for 942 requests. That means, only 58 requests received no suggestions. The graph can be seen in the Figure 5.5.

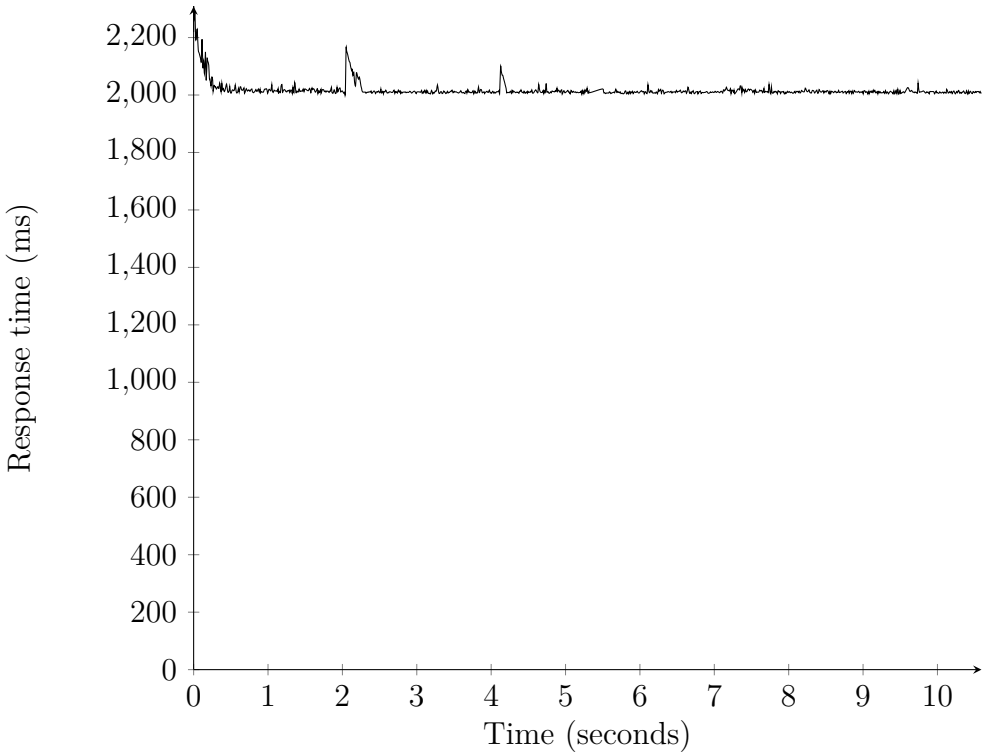

Figure 5.5: Load testing – worst case across multiple projects

5.4.3 Real-world Simulation

This test case tries to simulate the real-world usage. It contains all the simple prefix queries specified in the Section 5.4.1 (labeled 1-6), the worst case specified in the Section 5.4.2 (labeled 7), and the following queries (lone | signifies the caret position):

8. `". p|"` in *full* field.
9. `type:java full:p|`
10. `type:java full:public path:p|`
11. `type:c full:/|[aA].*/`
12. `type:java full:/|[aA].*/`
13. `public . v|` in *full* field.
14. `(org || buf) && |` in *defs* field.
15. `|a?` in *full* field.

16. `[a TO b|]` in *full* field.

17. `a*t|` in *full* field.

The requests specified all the 22 projects at once. 600 requests were sent during a 30 second period. Figure 5.6 shows how many requests ended successfully and how many with partial results. None of the requests returned an empty result. The times of the specific queries can be seen in the Figure 5.7. It can be noted that the worst query was not able to successfully finish within the specified time. Therefore, it influenced all the queries that followed.

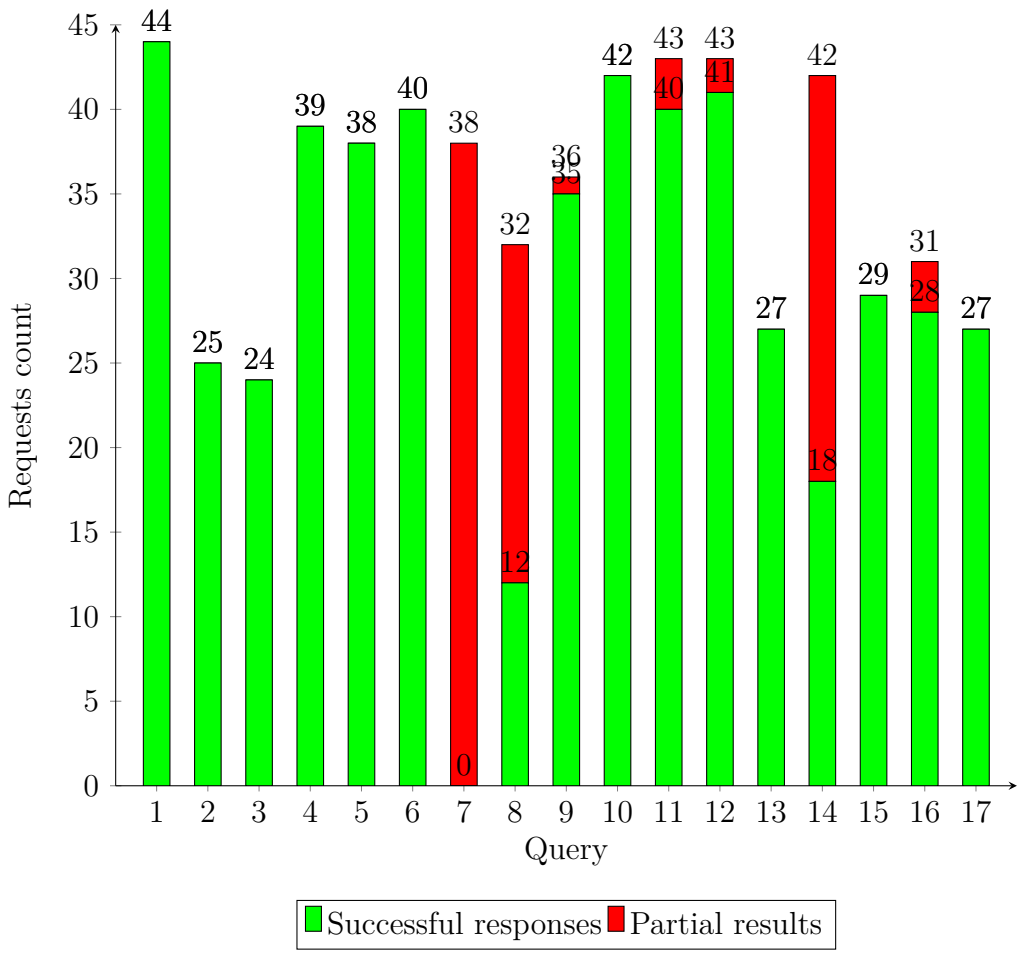

Figure 5.6: Comparison of successful and partial result counts in the real-world load test

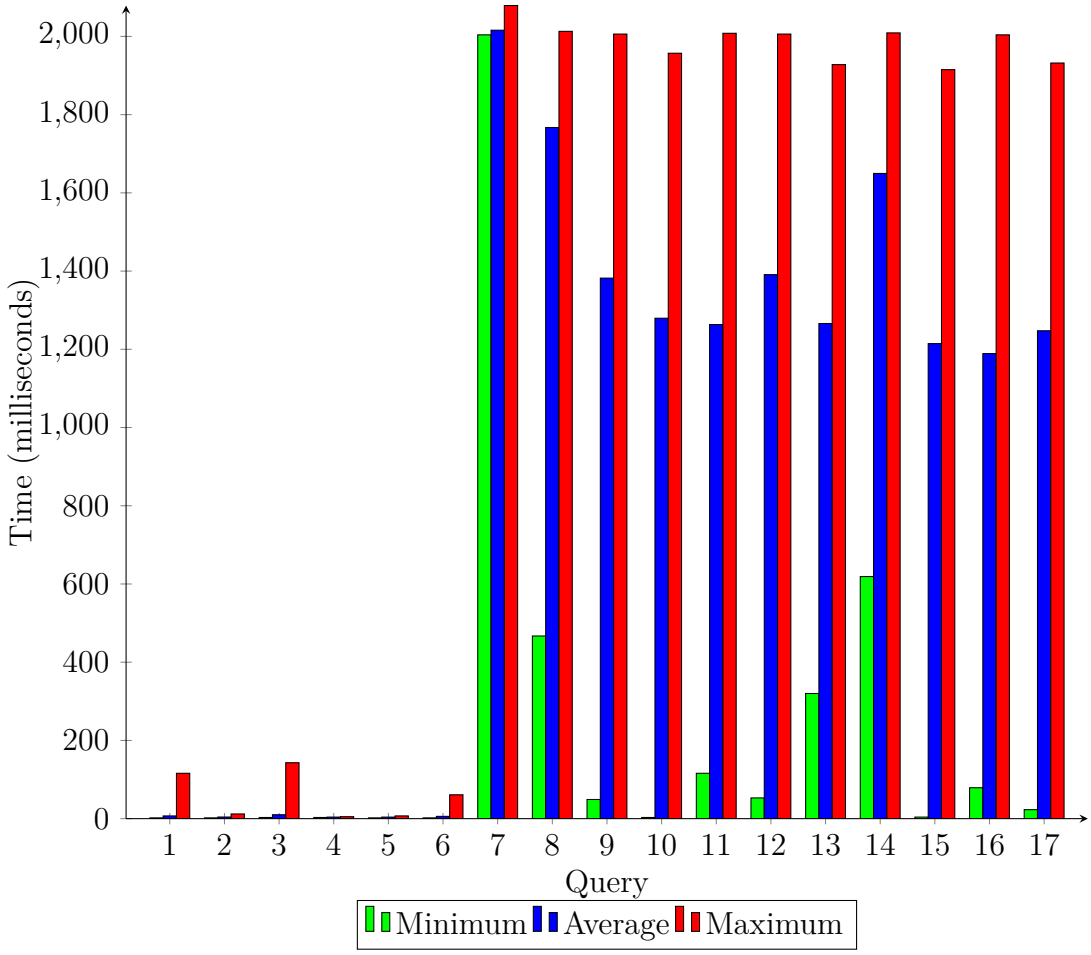

Figure 5.7: Times for specific queries

Conclusion

The target of the thesis was to implement an autocomplete functionality for the OpenGrok search engine. The implemented solution enabled this functionality for all the basic query types supported by OpenGrok. These query types were prefix queries, wildcard queries, regexp queries, range queries, fuzzy queries, and phrase queries. The implemented solution provides different suggestions according to the currently selected projects. Based on the data size and number of projects that are searched, the suggestions might take a long time to compute. Therefore, if the suggester is not able to return the results in a specified time, only partial results are returned.

The suggestions also provide correct results for multiple Lucene fields. That means, multiple data structures which store different information need to be maintained for every project in the OpenGrok instance.

To improve the suggestion quality, the most popular completion was used. The suggester remembers the search counts for every term and promotes the terms with the highest search counts. If no search data are available, the suggester uses the underlying statistics of the document set. It is also possible to initialize or update this data by using the API provided to administrators.

The solution provides a possibility to configure many aspects of the suggester to truly satisfy the users' needs. Among others, it is possible to specify minimum characters needed for suggestions to appear or to completely disable the suggestions.

There is no need to explicitly enable or configure the already existing OpenGrok instances. Only an upgrade to the version with the implemented suggester functionality is needed.

The existing solution can be improved and extended in many ways. First of all, optimizations for better suggestions speed should be implemented to provide better user experience. One of the optimizations that might be implemented is to use disk access for phrase queries with only a few documents which might increase their performance by an order of magnitude.

Secondly, suggestions might be improved to take into account even more factors and not just popularity and term or document frequency. Personalized suggestions for users is a very strong candidate for an extension. It is already implemented by Google and other search engines. Furthermore, error toleration is a very intriguing possibility for improvement. To err is human and to provide suggestions for a query with a small mistake might be an invaluable feature not only for the users on the mobile devices. Moreover, it might make sense to also make the suggester time-sensitive. In other words, the suggester could provide different suggestions based on the actual time or day.

Last but not least, suggestions could be more sophisticated if the suggester were to have a direct support for complex queries. For instance, it could boost different terms based on the already input data or it could recognize a pattern and suggest the rest of the complex query.

Bibliography

Ben Alex. LMDB benchmarks. https://github.com/lmdbjava/benchmarks/blob/master/results/20160710/README.md. Accessed: 2018-7-11.

Apache Software Foundation. Apache license 2.0. http://www.apache.org/licenses/LICENSE-2.0, 2004.

Ziv Bar-yossef and Naama Kraus. Context-sensitive query auto-completion, 2011.

H. Bast, Christian W. Mortensen, and Ingmar Weber. Output-sensitive autocompletion search. In *IN 13TH SYMPOSIUM ON STRING PROCESSING AND INFORMATION RETRIEVAL (SPIRE'06*, pages 150–162, 2006.

Arto Bendiken. The unlicense. http://unlicense.org/UNLICENSE, 2010.

Fei Cai, Shangsong Liang, and Maarten De Rijke. Time-sensitive personalized query auto-completion.

Surajit Chaudhuri and Raghav Kaushik. Extending autocompletion to tolerate errors. In *In SIGMOD Conference. 707–718*, 2009.

Russ Cox. Regular expression matching with a trigram index. https://swtch.com/~rsc/regexp/regexp4.html, 2012.

Free Software Foundation. GNU Lesser General Public License v3.0. https://www.gnu.org/licenses/lgpl-3.0.txt, 2007.

Vladimír Kotal and Adam Hornáček. OpenGrok Web Services Wiki. https://github.com/oracle/opengrok/wiki/Web-services. Accessed: 2018-7-17.

Greg Linden. Marissa Mayer at Web 2.0. http://glinden.blogspot.com/2006/11/marissa-mayer-at-web-20.html. Accessed: 2018-7-14.

Michael McCandless, Erik Hatcher, and Otis Gospodnetic. *Lucene in Action*. Manning Publications, 2 edition, 7 2010. ISBN 1-933988-17-7.

Stoyan Mihov and Denis Maurel. Direct construction of minimal acyclic subsequential transducers, 2001.

R. Miller. Response time in man-computer conversational transactions. In *AFIPS Fall Joint Computer Conference*, 1968.

Mehryar Mohri. Weighted finite-state transducer algorithms an overview. 2004.

Mehryar Mohri and Michael Riley. An efficient algorithm for the n-best-strings problem. In *In Proceedings of the International Conference on Spoken Language Processing 2002 (ICSLP '02*, 2002.

Michael Stapelberg. Debian code search. https://codesearch.debian.net/research/bsc-thesis.pdf, 12 2012.

Kryštof Tulinger. Authorization layer for the opengrok search engine. https://dspace.cvut.cz/handle/10467/69684, 2017.

List of Figures

List of Abbreviations

API Application Programming Interface – *set of methods for communication between various components*

CDDL Common Development and Distribution License – *open-source software license*

CEO Chief Executing Officer – *the top-ranking executive in a company*

CPU Central Processing Unit – *core computer component which performs instructions*

FST Finite State Transducer – *finite state automaton which can produce output*

HTML Hypertext Markup Language – *language for creating web pages*

IDE Integrated Development Environment – *application for software development*

IDF Inverse Document Frequency – *measurement of how important a term is in a document set*

JMX Java Management Extensions – *Java technology for managing resources*

JSON JavaScript Object Notation – *lightweight data-interchange format*

LDAP Lightweight Directory Access Protocol – *application protocol for accessing and maintaining distributed directory information services over network*

RCS Revision Control System – *version control system*

REST Representational State Transfer – *architectural style for creating web services*

TST Ternary Search Tree – *tree data structure*

WAR Web Application Resource – *file format for distributing Java web applications*

WFST Weighted Finite State Transducer – *FST with weights on arcs*

XML Extensible Markup Language – *markup language that defines a set of rules for encoding documents*

A. Benchmarks

Benchmarks are available at `https://github.com/Orviss/thesis_benchmarks`. *README.md* in the specified project contains instructions on how to run the benchmarks. All of the benchmarks were run on Apple Macbook Pro (15-inch, 2017) with the following configuration:

- **CPU** – Intel i7-7700HQ[1]
- **Memory** – 16GB of 2133MHz LPDDR3 onboard memory
- **Storage** – 256GB PCIe-based onboard SSD
- **Java version** – 1.8.0_162
- **Operating system** – macOS 10.13.5

More specific details about the machine can be found at `https://support.apple.com/kb/SP756?locale=en_US`.

Benchmarks did not run in an isolated environment; therefore, the results might have been influenced by the underlying operating system or other processes.

[1] `https://www.intel.com/content/www/us/en/products/processors/core/i7-processors/i7-7700hq.html`

B. Accepted Changes to OpenGrok

This chapter contains a list of accepted pull requests to the main OpenGrok's repository on GitHub. Only the changes concerning the focus of this work are listed. Accepted changes:

- Update jQuery UI to 1.12.1
  - Update of jQuery UI to contain the Autocomplete module.
  - `https://github.com/oracle/opengrok/pull/2132`
- Suggester
  - Suggester support for OpenGrok project.
  - `https://github.com/oracle/opengrok/pull/2212`

C. Content of the Attached File

Attached *.zip* file contains the following files:

- `javadoc` ..... generated documentation for OpenGrok project
- `opengrok` . OpenGrok source files including the source code for this thesis