GitHub Java code Search Engine

Alan Ihre
Kungliga Tekniska Högskolan
Stockholm, Sweden
aihre@kth.se

Alessio Galatolo
Kungliga Tekniska Högskolan
Stockholm, Sweden
galatolo@kth.se

André Silva
Kungliga Tekniska Högskolan
Stockholm, Sweden
andreans@kth.se

Anna Sánchez
Kungliga Tekniska Högskolan
Stockholm, Sweden
annase2@kth.se

April 12, 2020

Abstract

In this project report we present an end-to-end approach to information gathering in the domain of Java methods with unseen information granularity. We devise, implement and evaluate a search engine, called **GJSE**, short for ***G**ithub **J**ava code **S**earch **E**ngine*, that indexes Java methods and makes them available through structured queries via a user-friendly website. We evaluate this approach by crawling *GitHub*, indexing 408920 methods from 94 distinct projects, measuring query response times as well as performing a user survey to measure the relevance of search results. We find that our search engine is able to reach high performance both in terms of query time and relevance of results according to state-of-the-art measures.

1 Introduction

The exponential growth of open-source projects is enlarging the available code online [1]. This phenomenon opens an exciting opportunity for developers to extract information that would otherwise be expensive to find or create. Nevertheless, most of the tools used today for searching code, such as *GitHub*[2], have plenty of room for improvement. For instance, in the mentioned platform, the filters that can be applied to the search query only consist of filtering by coding language or other platform specific filters. We claim that a tool where the developer can search code with higher granularity, can more adequately solve her problem.

This project presents an innovative and robust solution to this need. Due to a simple user interface, the user can specify the domain-specific requirements of a search need with fine-grained parameters. In this first iteration of the project, we focus on Java code, as this is currently one of the most commonly used programming language[3]. Furthermore, method retrieval has been identified as the most common search need[4]. Our approach is, nonetheless, easily extensible to other programming languages and programming languages constructs, such as classes. We allow the developer to search for the required method by choosing the method name, return type, argument type among other useful filters.

Figure 1 provides an overview of our solution, which is comprised of 3 main units. First, a **Scanner** that, through *GitHub*'s API[5], crawls and identifies relevant files from code repositories. For each code repository, an **Indexer** is spawned. The indexer parses each relevant file and a new document, containing all relevant information, is posted to the *Elasticsearch*[6] cluster. In the final step, a website is made available to the client, the end-user of our solution, which simplifies the process of querying *Elasticsearch*.

The implementation of our solution, as well as instructions on how to run and replicate the experiments described in section 4, can be found under the following link `https://github.com/andre15silva/DD2476`.

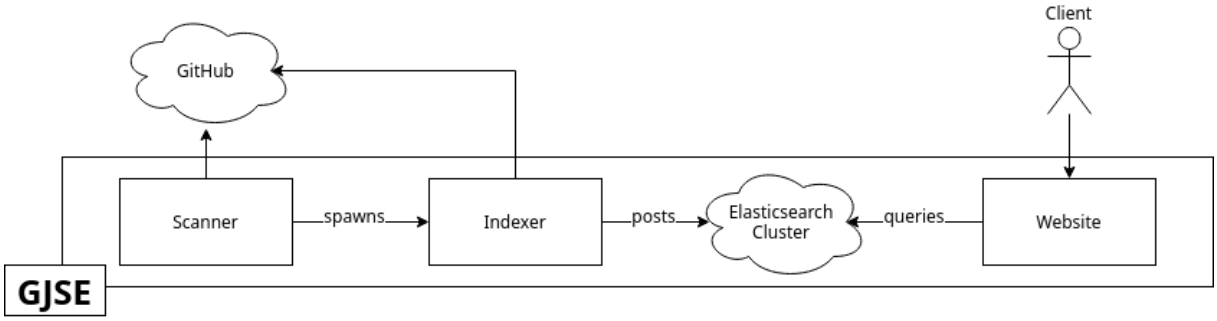


Figure 1: Overview of the architecture of *GitHub* Java code Search Engine

2 Related Work

Searching code snippets is not a new topic in the field of information retrieval. *Sourcerer*[7] leverages fine-grained structural information from code to store documents in a relational database, enabling both ranked search through a technique similar to *PageRank*[8] and structured queries. *CodeHow*[9] expands the strict structural query language into a natural language based one. *Portfolio*[10] focuses on the retrieval of functions and their usages, providing a visualization of chained usages. *CodeGenie*[11][12] builds on top of *Sourcerer*, taking test cases as queries. Sirres et al.[13] augment queries with information from *StackOverflow*. *FaCoY*[14] expands code queries with other tokens that might be relevant in the desired implementation. *Prequel*[15] enables searches in the commit history of a project. Urma et al.[16] allow for queries with graph databases.

Several online platforms exist or have existed. *Google Code Search* and *Koders* were code search engines that indexed open source projects. *Codase* was a code search engine that allows structured queries in several programming languages. *Krugle*[17] is company that deploys an open code search engine that allows textual search. *GitHub*[2] has a similar feature for the projects hosted in its platform. *SourceGraph*[18] is a startup whose product allows developers to search code repositories, including code navigation tools and a structural search model.

3 Methods

In section 1 we presented a brief overview of our approach. Now, we present a detailed explanation of each module.

3.1 Scanner

The scanner is a *Python* program that acts as a web crawler. Its objective is to crawl *GitHub* and identify relevant *Java* source code files that contain methods to be indexed later. It starts by obtaining a list of likely relevant repositories, with configurable length, retrieved through *GitHub*'s API[5]. As resources are limited, we employ effort in filtering repositories by visiting only repositories that contain code in *Java* and in a descending order according to the repository's stars.

Once the list of repository *seeds* is collected, each repository needs to be recursively visited, extracting the URLs of the Java files. Again, this step can easily be done with *GitHub*'s API. As a result, for each repository, a list of URLs of the Java files is obtained. Afterwards, for each repository, an indexer is spawned.

One important aspect of the scanner is that the amount of repositories that can be crawled per unit of time is limited by *GitHub*'s API. The rate limit is 1000 requests per hour per token. To circumvent this limitation, we employ a circular buffer of personal tokens, in an attempt of never running out of API calls.

3.2 Indexer

The indexer is the *Java* program responsible for indexing the code. Once launched by the scanner, it iterates over all identified files. Since this step requires retrieving the actual file contents, we also employ the same strategy for circumventing the rate limit explained in 3.1.

After retrieving a file, the indexer decodes and makes it available to the parsing step. In this step, the meta-programming library *Spoon*[19] is used. A meta-model is built, from which we retrieve all the information necessary for the indexing step.

In the indexing step we construct a POST request that contains the information retrieved in the parsing step. Table 1 contains an extensive listing of the information each indexed method contains.

An important note is that the program stores to disk only one file at a time to be immediately processed by *Spoon*. The filename is arbitrary and depends on the current process id, so as to allow many instances of the indexer to run in parallel. In fact, the only shared resource is the *GitHub* personal access tokens file, which is read only. The only bottleneck to full-on parallelization is then the *GitHub* rate limit for API calls.

3.3 User Interface

The user interface of the search engine provides the users with a graphical interface to interact with the search engine; *Elasticsearch*. This user interface is a web interface written in *React.js* and connects directly via HTTP to the REST API of the *Elasticsearch* cluster for querying the database.

Users are given two options in the user interface for sending queries to Elasticsearch, a basic mode and an advanced mode. In the basic mode the user can enter values for each field of the methods stored in the database. These input fields allows for *Elasticsearch* query expressions which are described in detail in the section on *Query Language*. The values of each input field are combined to a single query string and sent to the *Elasticsearch* cluster. The advanced mode has just one input field where the user can enter the entire query sent to the *Elasticsearch* cluster.

Field identifier	Meaning
repository	The repository identifier
className	Name of the class containing the method
fileURL	GitHub permalink to the source code file
returnType	Return type of the method
name	Method identifier
file	Name of the source code file
javaDoc	Documentation comment about the method
lineNumber	Line where the method declaration begins
visibility	Visibility modifier
modifiers	Extensive list of method modifiers
arguments	Extensive list of method arguments including name and type
thrown	Extensive list of declared exceptions
annotations	Extensive list of method annotations
preview	10 line preview of the method

Table 1: Schema of the method documents

Pictures of the basic and advanced query mode are shown in Figure 2 where the same query is entered via the two form types.

Search results are displayed to the user in a list with a limit of showing up to 100 results. Details of each method as well as the 10 first lines of the method code are displayed, as show in Figure 3.

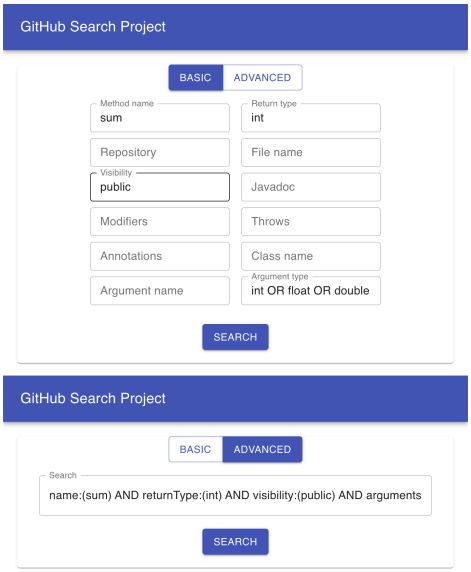


Figure 2: Example of the basic and advanced search interface for the same query

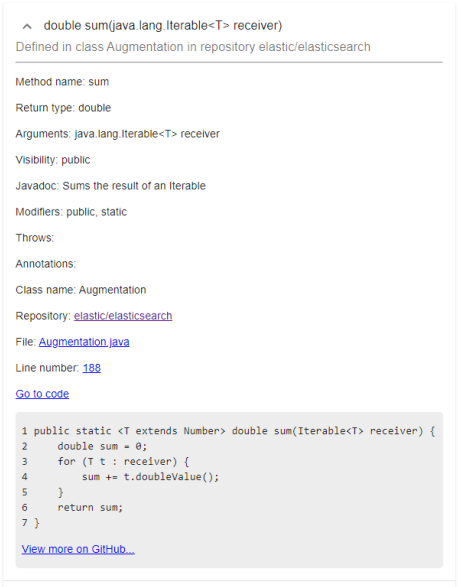


Figure 3: Example of search results

3.4 Query Language

The query language which users can use to query the search engine via the user interface is the query language provided by *Elasticsearch*. That is, the queries inputted by the user in the user interface are directly relayed to *Elasticsearch* as a "*Query string query*" in *Elasticsearch* terms. The provided query syntax is quite extensive and has too many functionalities to describe them all here. Therefore, a subset of them will be discussed here.

The query syntax[6] allows for specifying fields by writing the field name, followed by a colon, followed by the field value like in this example: `status:active`. If a field is not specified, all fields are searched for the query term. The query syntax also allows for wildcards and regular expressions to be used in the queries, enabling users to express a wide range of queries. There is also support for Boolean operators in form of `AND`, `OR` and `NOT`, as well as parentheses that can be used when multiple Boolean operators are considered. shows an example query written in this syntax.

```
name:(sum) AND returnType:(int) AND visibility:(public) AND arguments.type:(int OR float
    OR double)
```

Listing 1: Example query using *Elasticsearch*'s query syntax

4 Experiments

In this section we describe our evaluation of GJSE, namely the data we used and its collection process, the design of our experiments to measure query execution time and search result relevance, as well as present and discuss the corresponding results.

4.1 Data collection

To simulate an *Elasticsearch* cluster, a single-node cluster was launched, using the official docker image.

An instance of the scanner was ran for approximately 8 hours, scanning in total 94 repositories which yielded 408920 indexed methods. Additionally, a snapshot of the index was taken every hour and in the end of the process. Even though several personal access tokens were used during this run, the process was still suspended twice, having been resumed when more API calls were available. With API calls being the major limiting factor of the speed of our indexing, we decided not to evaluate the indexing speed. All experiments use this data as a basis.

The collected data is made available in the release section of the project repository, with instruction on how to replicate the experimental infrastructure under the directory `backend`.

4.2 Search result relevance

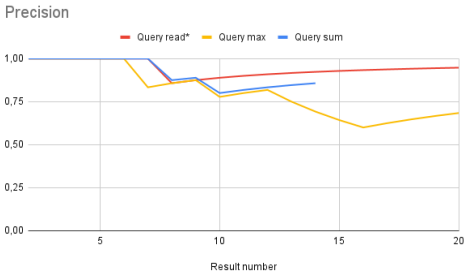
For an emerging search engine, an important experiment to make is to check the relevance of the result. For a predetermined query this can be done using multiple metrics such as precision, recall, mean average precision and the discounted cumulative gain (DCG). The precision is the percentage of relevant methods over the retrieved methods, whereas the recall measures the percentage of relevant retrieved methods over all the relevant methods of the search engine. Another metric that is highly important is the DCG, as it gives an idea of the performance of the ranking, penalizing relevant methods appearing in low positions.

All these offline metrics are obtained from the judgement of experts that score the relevance of a certain search result. For this reason, we employ a survey to be answered by users familiarized

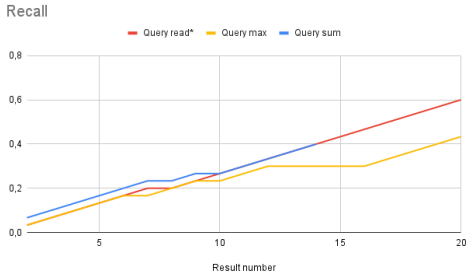
Query id	Method name	Return type	Argument type
Query sum	sum	-	-
*Query read**	read*	int	-
*Query max**	max*	int	int

Table 2: Selected queries for the expert survey

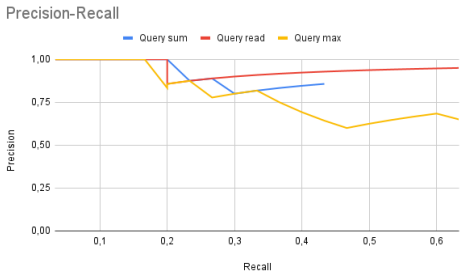
with the domain, where we propose three queries and for each, we report up to 20 results from our search engine. Both the survey and its answers can be found in the published repository under the directory `survey`. Table 2 shows the selected queries we used for this purpose.



(a) Precision @ k



(b) Recall @ k



(c) Precision-Recall

Figure 4: Precision and Recall

We collected, in total, the answers of 26 experts in Java. Each expert ranked each result for each query according to a relevance scale from 0 to 3, where: 0 means *Irrelevant method*; 1 means *Marginally relevant method*; 2 means *Fairly relevant method*; and 3 means *Highly relevant method*. To aggregate the different answers into a single relevance score for each search result presented we take the arithmetic mean over all answers.

Figure 4a shows the plot of the precision at k results for each query. It can be observed that the precision decreases as the number of methods retrieved (k) increases for the three queries. Nevertheless, the precision in the queries sum and read* is high compared to the precision in the max* query. The explanation behind this behavior is that the use of the wildcard character leads to the retrieval of a wide range of non-relevant methods such as "maxProductAfterCutting", which is a too specific method. Observing the same plot, it can be seen that the 5 first methods are relevant for all the queries, confirming the success of the search engine.

Figure 4b shows the plot of the recall at k results for each query. For the purpose of these calculations, we assume that the total amount of relevant methods in the system is 30. For all three queries, the recall is increasing as the number of methods retrieved is higher. The query

read* yields the best results, as it is the one that achieves better recall and it is almost a straight line (that means that almost all the methods retrieved are relevant).

For the precision-recall plot (Figure 4c), it can be observed the trade-off between the precision (y-axis) and the recall (x-axis) for different thresholds. From the plot we can see that the overall system has high precision but low recall. That means that it is returning too few but relevant results.

Figure 5 plots the mean average precision, MAP, across the queries at k. This metric is computed averaging the precision scores for each query. It is more stable and aims to capture the overall performance of a system. The results show that the user only has approximately a 25% on average overhead when trying to find relevant methods.

Figure 6 plots the normalized discounted cumulative gain (nDCG) at k. This metric is a measure that gives an average performance of a ranking algorithm. In an ideal system, where all the relevant methods are listed in the top positions, the nDCG is 1. In the plot, we can see that the ranking algorithm performs well in all the studied queries.

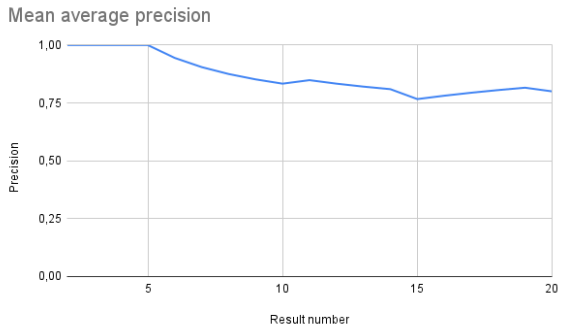


Figure 5: MAP

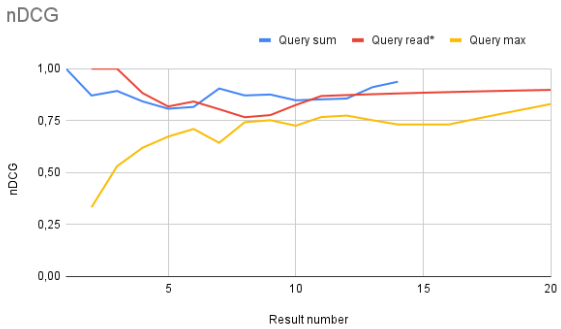


Figure 6: nDCG

4.3 Query execution time

The time it takes to execute a query is relevant to the user of the system. It is essential that the user can get search results in a swiftly manner even when the database size or the query result is large. To measure our search engine performance, we measured the time it takes for a set of queries for various database sizes. The query execution time is then averaged over 10 samples to avoid any outlier values affecting the result too much.

The method for evaluating the query execution time is as follows:

1. Load each snapshot into *Elasticsearch* and count the number of indexed methods in the database. These snapshots were taken at different times, as explained in 4.1, and as such vary in size.
2. Execute a set of queries 10 times each and record the query execution time and search result size.
3. Repeat step 2 until there are no more snapshots.

The query execution time we used is the one returned directly by *Elasticsearch*. However, this metric does not include the time it takes for the query and query result to be sent over the network to and from *Elasticsearch*.

In addition to the query execution time, the size of the returned result is recorded. This metric provides insights when studying the query execution time as the query execution time is likely affected by the query result size.

Table 3 shows the considered queries for this experiment, written in the advanced query mode syntax. Query #1 matches all entries in the database while the other ones are more specific. Both #2 and #3 are tested as the wildcard operator allows for inclusion of more results than the more specific query #2, which could result in longer query times. The last , #4, is even more specific but also includes a wildcard, which means that it tests both wildcard queries, conjunctive operations and matching towards specific query terms without wildcards (the arguments part).

#1	*
#2	*array*
#3	*array**
#4	*name:(array*) AND arguments.type:(int OR float)*

Table 3: Selected queries for the query execution time evaluation

The average query execution times in milliseconds and the query result sizes for each snapshot are shown in Figure 7 and 8. All of the curves in Figure 7 have similar levels of increase for larger database sizes, with the exception of query #3. The query execution time for query #3 increases much more rapidly, even though the query result size have a similar increase to the others. The query execution times for #2 and #4 actually decrease slightly for the largest database size tested. The query execution time for the query #1 which returns all results in the database remains more or less constant. It should be noted that the query result size graph uses a log scale for the y-axis displaying the result size.

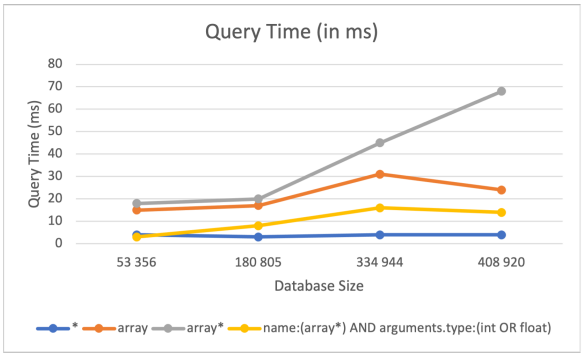


Figure 7: Query execution times.

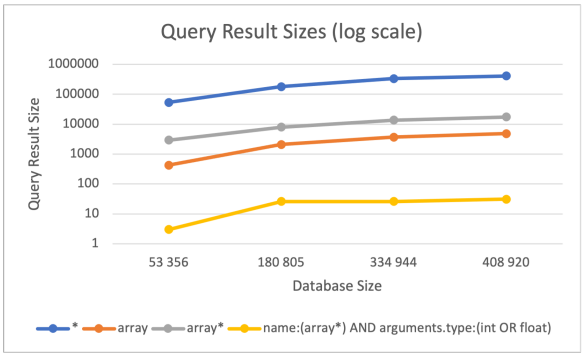


Figure 8: Query result sizes.

There are a couple of conclusions to be drawn from the resulting graphs for the query execution time and query result size.

First, the query time for the query #1, which matches all entries in the database, remains more or less constant as the database size increases. A probable reason for this is that while the query matches all entries in the database, the number of entries returned to the user interface is limited to 100. That is, the query engine stops the search after it has matched 100 entries with the query. As the size of the first snapshot tested is larger than 100 entries the search time for the all capture query will remain more or less the same as for the first tested snapshot.

Second, the query execution time increases as the database grows even though it seems to slightly decrease for the largest database size for two of the queries. Query #3 stands out from the others as its query execution time increases much more than for the other queries as the database size grows. One explanation for this can be that the size of the result for that query is much larger compared to the other queries, with the exception of query #2 which is closer in size. However, query #2 does not include a wildcard, which can explain why it has a significantly lower execution time compared to #3.

4.4 Threats to validity

In this section we present aspects that should be taken into consideration, and that could harm or invalidate the presented results.

4.4.1 Data collected

Even though we collected data for approximately 8 hours, 408920 methods is considerably far from the dimension of just *GitHub* alone. Furthermore, we only indexed methods from less than 100 projects. Both these aspects may mean we considered a skewed dataset, so results should take this into consideration.

4.4.2 Lack of comparison

Even though we computed several metrics, we did not compare them with previous work. This happens due to either a lack of available data to compare with, or due to lack of comparable approaches. Because of this, we cannot assess if our approach represents an improvement of the state-of-the-art in said metrics.

5 Conclusions

In this report, we have presented and proposed an end-to-end code search engine, *Github Java code Search Engine*. By using a user-friendly and adding more granularity in the search filters, the proposed solution solves the problems present by the current state-of-the-art. The experimental results both in terms of query time and relevance metrics show the robustness and viability of the engine, which can easily be extended to other coding languages and filters.

References

1. DESHPANDE, Amit; RIEHLE, Dirk. The Total Growth of Open Source. In: 2008, vol. 275. ISBN 978-0-387-09683-4. Available from DOI: 10.1007/978-0-387-09684-1_16.
2. *GitHub*. [N.d.]. Available also from: https://github.com/search.
3. *TIOBE Index*. [N.d.]. Available also from: https://www.tiobe.com/tiobe-index/.
4. SIM, Susan Elliott; CLARKE, Charles LA; HOLT, Richard C. Archetypal source code searches: A survey of software developers and maintainers. In: *Proceedings. 6th International Workshop on Program Comprehension. IWPC'98 (Cat. No. 98TB100242)*. 1998, pp. 180–187.
5. *GitHub's API*. [N.d.]. Available also from: https://docs.github.com/en/rest.
6. URMA, Raoul-Gabriel; MYCROFT, Alan. *Elasticsearch*. [N.d.]. Available also from: https://www.elastic.co/guide/en/elasticsearch/reference/current/query-dsl-query-string-query.html#query-string-syntax.

7. BAJRACHARYA, Sushil; NGO, Trung; LINSTEAD, Erik; DOU, Yimeng; RIGOR, Paul; BALDI, Pierre; LOPES, Cristina. Sourcerer: a search engine for open source code supporting structure-based search. In: *Companion to the 21st ACM SIGPLAN symposium on Object-oriented programming systems, languages, and applications*. 2006, pp. 681–682.
8. PAGE, Lawrence; BRIN, Sergey; MOTWANI, Rajeev; WINOGRAD, Terry. *The PageRank citation ranking: Bringing order to the web.* 1999. Tech. rep. Stanford InfoLab.
9. LV, Fei; ZHANG, Hongyu; LOU, Jian-guang; WANG, Shaowei; ZHANG, Dongmei; ZHAO, Jianjun. Codehow: Effective code search based on api understanding and extended boolean model (e). In: *2015 30th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. 2015, pp. 260–270.
10. MCMILLAN, Collin; GRECHANIK, Mark; POSHYVANYK, Denys; XIE, Qing; FU, Chen. Portfolio: finding relevant functions and their usage. In: *Proceedings of the 33rd International Conference on Software Engineering*. 2011, pp. 111–120.
11. LEMOS, Otávio A. L.; PAULA, Adriano C. de; ZANICHELLI, Felipe C.; LOPES, Cristina V. Thesaurus-Based Automatic Query Expansion for Interface-Driven Code Search. In: Hyderabad, India: Association for Computing Machinery, 2014, pp. 212–221. MSR 2014. ISBN 9781450328630. Available from DOI: 10.1145/2597073.2597087.
12. LEMOS, Otávio Augusto Lazzarini; BAJRACHARYA, Sushil Krishna; OSSHER, Joel; MORLA, Ricardo Santos; MASIERO, Paulo Cesar; BALDI, Pierre; LOPES, Cristina Videira. Codegenie: using test-cases to search and reuse source code. In: *Proceedings of the twenty-second IEEE/ACM international conference on Automated software engineering*. 2007, pp. 525–526.
13. SIRRES, Raphael; BISSYANDÉ, Tegawendé F.; KIM, Dongsun; LO, David; KLEIN, Jacques; KIM, Kisub; TRAON, Yves Le. Augmenting and Structuring User Queries to Support Efficient Free-Form Code Search. In: *Proceedings of the 40th International Conference on Software Engineering*. Gothenburg, Sweden: Association for Computing Machinery, 2018, p. 945. ICSE '18. ISBN 9781450356381. Available from DOI: 10.1145/3180155.3182513.
14. KIM, Kisub; KIM, Dongsun; BISSYANDÉ, Tegawendé F; CHOI, Eunjong; LI, Li; KLEIN, Jacques; TRAON, Yves Le. FaCoY: a code-to-code search engine. In: *Proceedings of the 40th International Conference on Software Engineering*. 2018, pp. 946–957.
15. LAWALL, Julia; LAMBERT, Quentin; MULLER, Gilles. Prequel: A patch-like query language for commit history search. 2016.
16. URMA, Raoul-Gabriel; MYCROFT, Alan. Source-code queries with graph databases—with application to programming language usage and evolution. *Science of Computer Programming*. 2015, vol. 97, pp. 127–134. ISSN 0167-6423. Available from DOI: https://doi.org/10.1016/j.scico.2013.11.010. Special Issue on New Ideas and Emerging Results in Understanding Software.
17. *Krugle Open Search*. [N.d.]. Available also from: https://opensearch.krugle.org/.
18. *Sourcegraph*. [N.d.]. Available also from: https://about.sourcegraph.com/.
19. PAWLAK, Renaud; MONPERRUS, Martin; PETITPREZ, Nicolas; NOGUERA, Carlos; SEINTURIER, Lionel. Spoon: A Library for Implementing Analyses and Transformations of Java Source Code. *Software: Practice and Experience*. 2015, vol. 46, pp. 1155–1179. Available from DOI: 10.1002/spe.2346.