SEMANTIC CODE SEARCH AND ANALYSIS

A THESIS IN
Computer Science

Presented to the Faculty of the University
of Missouri-Kansas City in partial fulfillment
of the requirements for the degree

MASTER OF SCIENCE

By

YASHWANTH RAO DANNAMANENI

B.TECH, Amrita Vishwa Vidyapeetham, 2012

Kansas City, Missouri

2014

SEMANTIC CODE SEARCH AND ANALYSIS

Yashwanth Rao Dannamaneni, Candidate for the Master of Science Degree

University of Missouri-Kansas City, 2014

ABSTRACT

As open source software repositories have been enormously growing, the high quality source codes have been widely available. A greater access to open source software also leads to an increase of software quality and reduces the overhead of software development. However, most of the available search engines are limited to lexical or code based searches and do not take semantics that underlie the source codes. Thus, object oriented (OO) principles, such as inheritance and composition, cannot be efficiently utilized for code search or analysis.

This thesis proposes a novel approach for searching source code using semantics and structure. This approach will allow users to analyze software systems in terms of code similarity. For this purpose, a semantic measurement, called CoSim, was designed based on OO programing models including Package, Class, Method and Interface. We accessed and extracted the source code from open source repositories like Github and converted them into Resource Description Framework (RDF) model. Using the measurement, we queried the source code with SPARQL Query Language and analyzed the systems. We carried out a pilot study for preliminary evaluation of seven different versions of Apache Hadoop systems in terms of their similarities. In addition, we compared the search outputs from our system with those by the Github Code Search. It was shown that our search engine provided more comprehensive and relevant information than the Github does. In addition, the proposed CoSim measurement precisely reflected the significant and evolutionary properties of the systems in the similarity comparison of Hadoop software systems.

APPROVAL PAGE

The faculty listed below, appointed by the Dean of the School of Computing and Engineering, have examined a thesis titled “Semantic Code Search and Analysis,” presented by Yashwanth Rao Dannamaneni, candidate for the Master of Science degree, and hereby certify that in their opinion, it is worthy of acceptance.

Supervisory Committee

Yugyung Lee, Ph.D., Committee Chair

School of Computing and Engineering

Praveen Rao, Ph.D., Committee Co-Chair

School of Computing and Engineering

Yongjie Zheng, Ph.D., Committee Member

School of Computing and Engineering

Table of Contents

ILLUSTRATIONS

TABLES

CHAPTER 1

INTRODUCTION

1.1 Motivation

A software programmer while working on an application will reuse of the existing source code. Reuse of the code is very important, as most applications are not completely novel. For reusing existing source-code, we need to search for existing, working code, which has the same user functionalities. With the popularity of open source software repositories [10] there is enormous high quality open source software code available. Reusing as much of this code as possible could improve the software development as most of the code that is used for this application is already been written for some other application and is available in an open source software repository. This makes programmers to collect raw source code from open source software repositories to use it in their own application as a reusable component, library, or simply an example [2].

However, most of this source code are very poorly organized and distributed among different open source software repositories. A very little amount of source code will be useful to reuse when ineffective search methodologies are used. While these systems are promising, they do not seem to leverage the various complex relations present in the code, and therefore have limited features and search performance. Many search engines as well as software development hosting services like Github provide search facility and most of these search engines only provide text or keyword based search hence limiting the search for keywords. So finding appropriate code fragments for a particular application with user needs is difficult and even may be irrelevant.

As of today, open source software repositories are enormous; there is a very difficult challenge to find relevant source code. For example, Github alone hosts more than 10 million repositories both public and private repositories [23], sourceforge hosts more than 300,000 open source projects [24].

Once you have the required source code a programmer needs to analyze or sometimes even reengineer the software system. This is a very time consuming and costly process to understand the source code behavior, organization and architecture of the software system [3]. As there are times where the searched code has different versions, as most open source software's are evolving. As there are different versions, there is a need for a method to check the similarities between the systems.

1.2 Problem Statement

As software repositories are growing enormously, there is a working, high quality code available to software programmers for reusing as a component, library in their application. As the source code for some applications is huge, analyzing these source codes is time consuming. There are solutions out there where we can search code and analyze code. However, all these solutions offer a keyword based search, which is not efficient when it comes to source code. Source code has structure in it, which means we have a structure that can be used for querying. How can we query for source code with the help of its structure? The solution must be scalable, performance driven and cost effective.

The application needs to be user friendly, which makes users to find the required source code easily, and consider structure of the source code for query processing.

1.3 Thesis Outline

This thesis focuses on the problems of existing open source-code search engines available. As most of the developing applications are not completely novel, so reusability of source code improves software development. As there are no efficient search techniques to search source code reusability of source code is still a very important problem in software development. This thesis aims on the importance of structure of source code for efficient searching. It replaces the keyword-based search with structure-based search. The principle here is to extract the structure from the source code and represent it as Resource Description Framework (RDF) [7][8] and query the extracted RDF dataset with the help of SPARQL[14].

The thesis also focuses on analyzing open source software systems, the principle here is to find similarity between two different software systems(may be different version) based on the structure of the source code. The RDF dataset extracted from the source code is input to the similarity model. This is useful to find similarities at each level of the source code. The similarity measurement found here would be used for semantic search for more comprehensive search results.

CHAPTER 2

BACKGROUND AND RELATED WORK

In this chapter, we will introduce the technologies used, the problems of lexical code search engines and the related work. It includes different searching approaches, comparison among existing source code search systems and semantic analysis of source code.

2.1 Background

In this section, we discuss some of the technologies that are used in the implementation of the Semantic Code Search and Analysis System.

Resource Description Framework (RDF) [8]: RDF is a standard model for data interchange on the web. RDF extends linking structure of the web to use URIs to name the relationship between things as well as the two ends of the link. RDF uses subject-predicate-object expressions.

- Triple: Each of the subject-predicate-object expression is a Triple.
- Subject: A subject denotes the resource.
- Predicate: A predicate denotes traits or aspects of the resource. It also expresses a relationship between subject and object.
- Object: An object denotes the instance of the resource.

For example “The sky has the blue color” in RDF is a triple. Here “The sky” is the subject, “has” is the predicate and “the blue color” is the object.

SPARQL [14]: SPARQL is an RDF query language able to retrieve and manipulate data stored in RDF format. SPARQL language specifies four different query variations for different purposes. Some of the query patterns useful are SELECT query, CONSTRUCT query, ASK query and DESCRIBE query.

- SELECT query is used to extract raw values from SPARQL endpoint.
- CONSTRUCT query is used to extract information from the SPARQL endpoint and transform the results in a table format.
- ASK query is used to provide a simple True/False result for a query on a SPARQL endpoint.
- DESCRIBE query is used to extract an RDF graph from the SPARQL endpoint, the contents of which is left to the endpoint to decide based on what the maintainer deems as useful information.

Apache Jena: As Jena [17] official site states, "Jena is a framework for building semantic applications. It provides a programmatic environment for RDF, RDFS, and OWL, SPARQL and includes a rule based inference engine". Apache Jena is developed in Java programming language. It provides API to write data to and get data from RDF graphs. It also provides ARQ engine, which runs SPARQL queries on RDF data. To query RDF data, a model will be created using RDF statements and SPARQL will execute on this model.

2.2 Problems of Lexical Code Search Engines

One of the first things programmers do when writing new code is find existing, working code with similar functionalities. As many of the applications written today are not completely novel, one could fetch a significant amount of existing code needed for the application from the open source software repository [1].

Unfortunately, the reuse of open source code is very little. There are several reasons for this. The first is that equivalent code is difficult to find as traditional search engines offers only keyword based search and the second is that the code found will rarely meet the user's requirements [1].

2.3 Related Work

Source code searching allows users to browse, search and retrieve source code from large software repositories. Existing source code searching systems use different methods for searching such as keyword based, tag based, Meta information based and structure based. There are different types of source code searching approaches.

SEARCH BY KEYWORD [11]: This is more like traditional search where the users can search for content using keywords; the search engine returns the documents, which has those keywords in it. This is a very good approach when it comes to text documents.

SEARCH BY TAG [20]: In this search approach, the users can search which keywords or tags and the search engine returns the documents or contents, which are associated with those keywords. This is a very good approach to content that can be categorized with the help of tags.

SEARCH BY META INFORMATION [9]: In this search approach, the search engine makes use of the Meta information to provide the search results.

SEARCH BY STRUCTURE OR MODEL [12][13]: In this search approach, the search engine takes the structure or model into consideration, the user can provide some structural information and the search engine returns with matching documents with the provided structural information.

There are many existing source code-searching systems; some of them are koders, codase, krugle, SparsJ search engine, meanpath. These search engine use keyword based search and some only consider structure of the source code into consideration. Many approaches for source code searching are available, searching the source code with the help

of metadata, or by using users input and applying transformations to map it with retrieved code into what the user asked for [1].

Some approaches also include extracting fine-grained structural information from the code. This information is useful to implement search forms that go beyond conventional keyword based searches [2]. There are approaches that take queries of the forms source object type and destination object type as input to suggest relevant method invocation sequences. They serve as solutions that yield the destination object type from source object type [6].

Typestate semantic code search [5] handles partial programs in the form of code snippets. Handling snippets allows us to consume code from various sources and extract possibly partial temporal specification from each snippet using a relatively precise static analysis tracking and for querying they use define a notion of relaxed inclusion matching a query against temporal specifications and their corresponding code snippets.

Table 2.1: Comparison of Different Source Code Search Engines

Search Engine	Query Type	Uses Structure	Search Used for
Google Code [11]	Tag based	No	Projects
Sourceforge [20][24]	Tag based	No	Projects
Github [9]	Keyword and meta information based search	No	repositories, source code
Codase [12]	structure based search	Yes	source code
Krugle [13]	Keyword and meta information based search	No	source code
SparsJ [19]	Keyword based	No	Classes
MeanPath [21]	Meta information	No	search websites

	based search		through its meta information

Understanding the source code for maintenance and reengineering is very difficult and challenging task [3]. LSA extracts the information needed for complete understanding of any part of the software system. LSA identifies the similarities between two software systems and determines how well such a method can be used to support aspects of program understanding, comprehension, and reengineering of software systems. LSA uses corpus based statistical method for representing words; this representation is useful to find similarity measure.

There are different types of software analysis.

SYSTEM LEVEL: At system level, one number as a measurement of every snapshot of a software system is generated. Lehman et al: studied such measurements and generalized eight laws for software evolution.

SUB-SYSTEM LEVEL: At subsystem level, the evolution study is performed on every subsystem of the software project to understand more details about the evolution of a software system. For example, Godfrey et al: study the evolution of lines of code of every subsystem of the Linux kernel.

FILE LEVEL: The evolution study at file level reports the evolution of every source file of a software project. For example, the evolution study would report evolutionary information such as le \a.c" is 5 lines less after the most recent change.

ENTITY-CODE LEVEL: At entity code level, every snapshot of every source code entity, such as function, is recorded. The differential analysis would generate results such as “CLASS A is changed, Function foo is modified”.

ABSTRACT SYNTAX TREE: At AST level, an AST is built for every snapshot of source code. The analysis made by tracking the changes in the AST.

CHAPTER 3

SEMANTIC CODE SEARCH AND ANALYSIS MODEL

In this chapter, we proposed a model for semantic search and analysis. It includes extracting features from source code and storing the features as a semantic network, semantic analysis of different software systems and semantic query of source code.

3.1 Overview

As there are large open source repositories that contains rich source code with different versions, we need an effective search engine, which is capable of searching source code by the structure and an analysis system, which provides detailed similarity information for the search results. We designed a similarity model to find the similarities between two software systems; we use Jaccard Similarity Index [15] principle to find similarity between two strings or predicates of a triple in RDF. We provide a new similarity measurement coSim that is applied to sematic query system, which queries the source code by its structure. The extracted structure from source code from different software versions are stored as RDF's in a datastore. Now we query this datastore for search results using SPARQL [14]. The Apache Jena [17] will process the query.

As there are different versions of the source code, semantic analysis provides the search results with the help of the coSim measurement for each version of source code available in the system. We use Apache Hadoop software system source code for evaluating Semantic Search and Semantic Analysis.

3.2 Data Model

The Kabbalah model [7] is the RDF model used. It defines resources and relationships among resources. A resource identifier represents every resource within the

model. Almost every resource stored within a model is an Identifier.

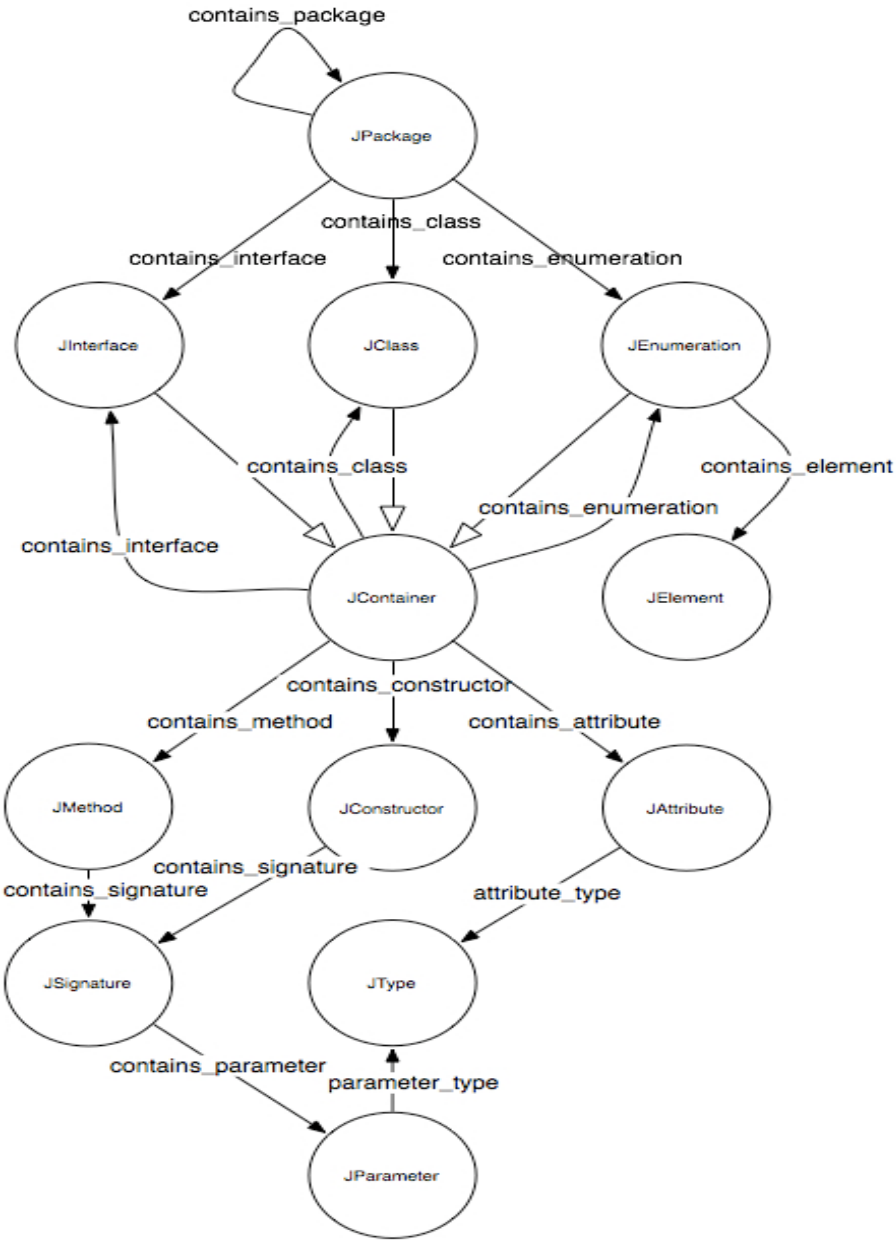

Figure 3.1 Kabblah Model [7]

The identifiers have a fixed structure that can be represented with the following *BNF* rules:

<IDENTIFIER> ::= (<PREFIX> '#')? <SECTIONS>;

<SECTIONS> ::= <SECTIONS> '.' <SECTION> | <SECTION>;

<SECTION> ::= (qualifier ':')? Fragment;

Every identifier can have a prefix that, if present, must end with '#', The identifier is composed of a sequence of sections separated by'.', every section can contain a qualifier and must define a fragment. The characters '#' '.' ':' are used to identify the different parts of the identifier, fir this reason they cannot be used neither in the prefix nor in qualifiers or fragments.

An example of valid identifiers are:

http://www.rdfcoder.org/2007/1.0#jpackage:org.apache.hadoop.mapreduce.jclass:Mapper

We create two different models for two different software systems using the above-mentioned Kabblah model. Below is a simple method declaration and its corresponding RDF.

The identifier used in this example is

#jpackage:javachat.network.message.jmethod:createHeloPacket

The identifier includes the package name “javachat.network.message” and the method name “createHeloPacket”.

Table 3.1: RDF example

Example	RDF
public packet createHeloPacket(String newName, String oldName) { }	<rdf:Description rdf:about="#jpackage:javachat.network.message.j method:creat eHeloPacket"> <j.0:contains_signature rdf:resource="#jpackage:javachat.network.message.jmethod: createHeloPacket. jsignature:_0"/> <j.0:has_visibility>public</j.0:has_visibility> <j.0:has_modifiers>4</j.0:has_modifiers> <rdfs:subClassOf rdf:resource="#JMethod"/> </rdf:Description>

3.3 Semantic Similarity

We developed a similarity model to find similarities between two different software systems. We take two different RDF datasets (two software systems) and give them as input to the similarity model. Now we create two models of different RDF datasets using Apache Jena. The similarity model calculates the similarity between two RDF nodes. The flow of operation is as shown in Figure 3.2.

The model uses the Jaccard similarity index to compute the similarity between two literals or predicates. The formulae for the same is as shown in Figure 3.3.

$$J(A,B) = \frac{|A \cap B|}{|A \cup B|}.$$

Figure 3.2 Jaccard Similarity Index [15]

Here A and B are two different software system RDF sets and J(A,B) is a similarity measurement between finite sample sets A,B.

The model also uses the Levenshtein string distance algorithm to calculate the distance between two strings. This is to calculate the uniqueness between two words in two statements of the same RDF node. The formulae to calculate Levenshtein distance as shown in Figure 3.4.

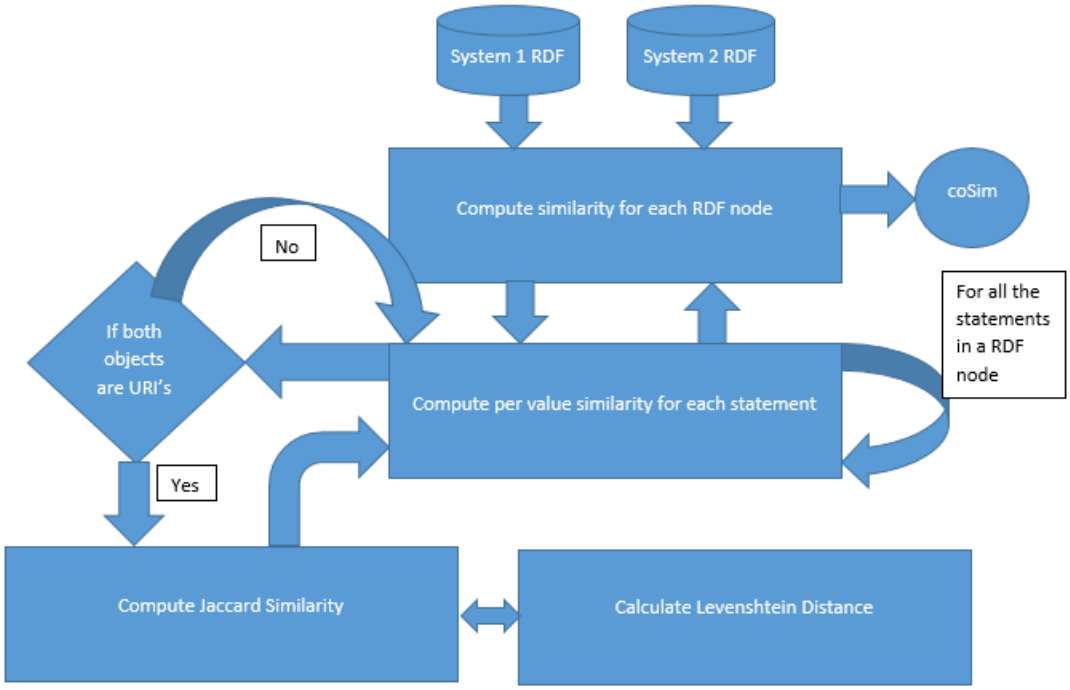

Figure 3.3 Similarity Computation

$$\operatorname{lev}_{a,b}(i,j) = \begin{cases} \max(i,j) & \text{if } \min(i,j) = 0, \\ \min \begin{cases} \operatorname{lev}_{a,b}(i-1,j) + 1 \\ \operatorname{lev}_{a,b}(i,j-1) + 1 \\ \operatorname{lev}_{a,b}(i-1,j-1) + 1_{(a_i \neq b_j)} \end{cases} & \text{otherwise.} \end{cases}$$

Where a, b are the strings or literals

i,j are the string positions

Figure 3.4 Levenshtein String Distance [18]

The three operations that take place in levenshteins string distance are insertion, deletion and substitution.

Definitions:

- Per value similarity(pvSim): pvSim is the similarity measurement for an individual statement of a RDF node.

- Similarity: Similarity is the similarity measurement for each RDF node.
- coSim: coSim is the similarity measurement for the whole system or for the given RDF.

The model includes several steps in computing the coSim similarity measurement. Each step is explained below in detail.

1. Similarity for RDF node: In this step, we select a RDF node depending on the level of user's choice (Package, Class, Method, Constructor, or Interface). Find all the similar statements associated with the RDF node. Let us take the

   "*jpackage:javachat.network.message.jmethod:createHeloPacket*"

   RDF node from the example shown in table 2.
2. Calculate pvSim for each statement: In this step, we select a statement and identify the literal in the statement. The list of statements for the RDF node

   "jpackage:javachat.network.message.jmethod:createHeloPacket" are

   <j.0:has_visibility>public</j.0:has_visibility>

   <j.0:has_modifiers>4</j.0:has_modifiers>

   These are the list of statements from version1. Now let us take statements from a different version for the same node.

   <j.0:has_visibility>private</j.0:has_visibility>

   <j.0:has_modifiers>4</j.0:has_modifiers>

   Now we calculate the pvSim for each individual statement.
3. Jaccard similarity for each statement : If both the statements are literals we use Jaccard similarity index to calculate the similarity score between the literals in a

statement. As Public and private are different words. The Jaccard similarity score for this statement would be 0.

For finding the difference between two words, we use the levenshtein's string distance principle to calculate distance between two words for more accurate word matching.

4. Calculate similarity for the RDF node : We use the formulae shown in figure 3.6 to calculate the similarity for each RDF node.
5. Calculate coSim for the entire system : We find the average of all the RDF node similarity to calculate coSim for the entire system. The coSim ranges from 0 to 1. Where 0 meaning no similarity between the systems and 1 meaning both the systems are identical.

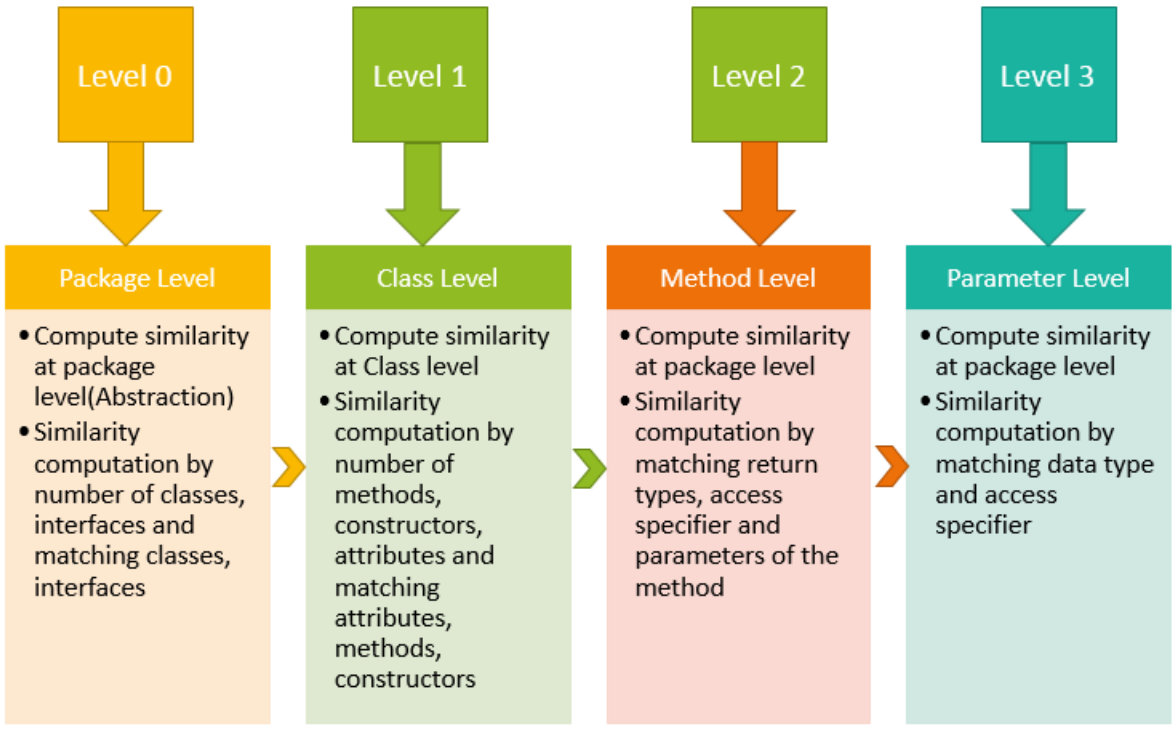

Figure 3.5 Scoring Levels

The model finds similarity for Package level, class level, method level and parameter level. First it goes in to package level then class level method level and then parameter level. Calculates the score for the parameter level and sums it up for the method level then class level and the package level as shown in Figure 3.5.

$$Similarity = pvSimSum / (pvSimSum + alpha * (object1\_unique) + beta * (object2\_unique))$$

where alpha=1,beta=1

pvSimSum is the Jaccard Similarity for the two statements

object1_unique is the number of unique objects for the RDF node n1 from Model m1

object2_unique is the number of unique objects for the RDF node n2 from Model m2

Figure 3.6 RDF node Similarity Formulae

The similarity model also provides the concept of levels, for improving the performance based on user's request. There are three levels (1, 2, and 3). Using the level 3 makes the comparisons until parameter level, level 2 for method level, level 1 for class level. So lesser the level less the comparisons but will affect the accuracy of the results.

Table 3.2: Algorithm to Compute Similarity

```
Compute_similarity(RDFNode r1, Model m1, RDFNode r2, Model m2, int level) {
 Select the RDF node based on the user choice of Level
 Get list of statements for both RDF nodes r1 and r2.
 for each statement in statement s1
 {
    for each statement in statement s2
    {
      pvSim=compute pvSimilarity(stm1,m1,stm2,m2)
    }
 }
 Aggregating all the pvSim gives Document similarity
}
```

Table 3.3: Algorithm to compute pvSimilarity

```
pvsimilarity(RDFNode r1, Model m1, RDFNode r2, Model m2, int level)
{
 Depending on the level compare the statements stm1 and stm2 from model m1,m2.
 If the stm1 is a value or predicate use levenshtein string distance to compute the string distance between two literals.
 Use Jaccard Index to find pvSim between two statements
}
```

3.4 Semantic Query

Semantic Code Search Engine provides different search features. The features provided by the search engine are *Package level search, Class level search, Method level search, Constructor level search and Interface level search*. The search engine also provides search capabilities, which can infer many more queries like "Find a Class which implements this Interface".

Generation of SPARQL queries is to query the RDF data store for packages, classes, methods, constructors or interfaces. According to the user's choice of search, the query generator makes decisions to make appropriate queries for querying. Here are the different SPARQL queries generated by the system.

- Search by Package: In this case, the user will search for packages with the help of package name. The SPARQL query generator to create a SPARQL query for searching packages in the dataset will use the package name. The result is a list of classes in the package.

Table 3.4: SPARQL Query for Search by Package

```
select  ?uri
       where
        {
         ?uri rdfs:subClassOf <http://www.rdfcoder.org/2007/1.0#"+package_name+">
        }
```

- Search by Class: In this case, the user will search for classes with the help of class name, extended class, and visibility of class, by implemented interface or by method in the class. The result will be a list of classes, matching the user's requirements.

Table 3.5: SPARQL Query for Search by Class

```
select ?class ?method ?impinterface ?extended_class ?visibility
where
{
    ?x j.0:contains_class ?class
              FILTER (REGEX(STR(?class), 'jclass:"+class_name +"', 'i'))
    OPTIONAL
    {
       {
            ?class j.0:contains_method ?method
             FILTER (REGEX(STR(?method), 'jmethod:"+contains_method +"', 'i'))

       }

       UNION

       {

        ?class j.0:implements_int ?implements_int

                    FILTER (REGEX(STR(?implements_int), '"+implements_int +"', 'i'))
       }
       UNION
       {
         ?class j.0:extends_class ?extends_class
                    FILTER (REGEX(STR(?extends_class), '"+extends_class +"', 'i'))
        }
        UNION
```

```
        {
         ?class j.0:has_visibility ?visibility
                        FILTER (REGEX(STR(?visibility), '"+visibility +"', 'i'))
        }
    }
}
```

- Search by Method: In this case, the user will search for methods with the help of method name, return type, visibility and exception thrown. A list of methods, which contained all or some of these, will be the result.

Table 3.6: SPARQL Query for Search by Method

```
select ?throws ?signature ?visibility
   where
   {
    ?x j.0:throws ?throws
                FILTER (REGEX(STR(?throws), "+exeception_name+", 'i'))
    ?x j.0:contains_signature ?signature
                FILTER (REGEX(STR(?signature),'jmethod:"+method_name+"', 'i'))
    ?x j.0:has_visibility ?visibility
                FILTER (REGEX(STR(?visibility), '"+visibility+"', 'i'))
}
```

- Search by Constructor: In this case, the user will search for constructor with the help of constructor name, visibility or by constructor parameter. A list of constructors with these parameters are the result.

Table 3.7: SPARQL Query for Search by Constructor

```
select ?constructor ?contains_parameter ?visibility
   where
   {
     ?x j.0:contains_constructor ?constructor
      FILTER (REGEX(STR(?constructor), 'jconstructor:"+constructor_name +"', 'i'))
     ?x j.0:contains_parameter ?parameter
      FILTER (REGEX(STR(?parameter),'jparameter:"+parameter_name+"', 'i'))
```

```
    ?x j.0:has_visibility ?visibility
    FILTER (REGEX(STR(?visibility), '"+visibility+"', 'i'))
  }
```

- Search by Interface: In this case, the user will search for Interface with the help of Interface name, method or by attribute. A list of interfaces will be the result.

Table 3.8: SPARQL Query for Search by Interface

```
Select ?interface ?method ?attribute
   where
   {
     ?x j.0:contains_interface ?interface
     FILTER (REGEX(STR(?interface), 'jinterface:"+interface_name +"', 'i'))
     ?x j.0:contains_method ?method
     FILTER (REGEX(STR(?method),'jmethod:"+method_name+"', 'i'))
     ?x j.0:contains_attribute ?attribute
     FILTER (REGEX(STR(?attribute), '"+attribute_name+"', 'i'))
   }
```

In this chapter, we discussed Semantic Similarity and Semantic Query model, which includes RDF model, Similarity measurement coSim and semantic query system. In the next chapter, we will discuss implementation aspects of our system.

CHAPTER 4

SEMANTIC CODE SEARCH AND ANALYSIS IMPLEMENTATION

In this chapter, we will discuss the detailed implementation of our system that includes architecture of our systems and the various components involved in the system.

4.1 Architecture

The architecture is based on semantic querying, semantic similarity methodologies. It is as shown in Figure 4.1. The diagram provides the source code parser, different query types the system provides, query generator, query execution and the similarity model.

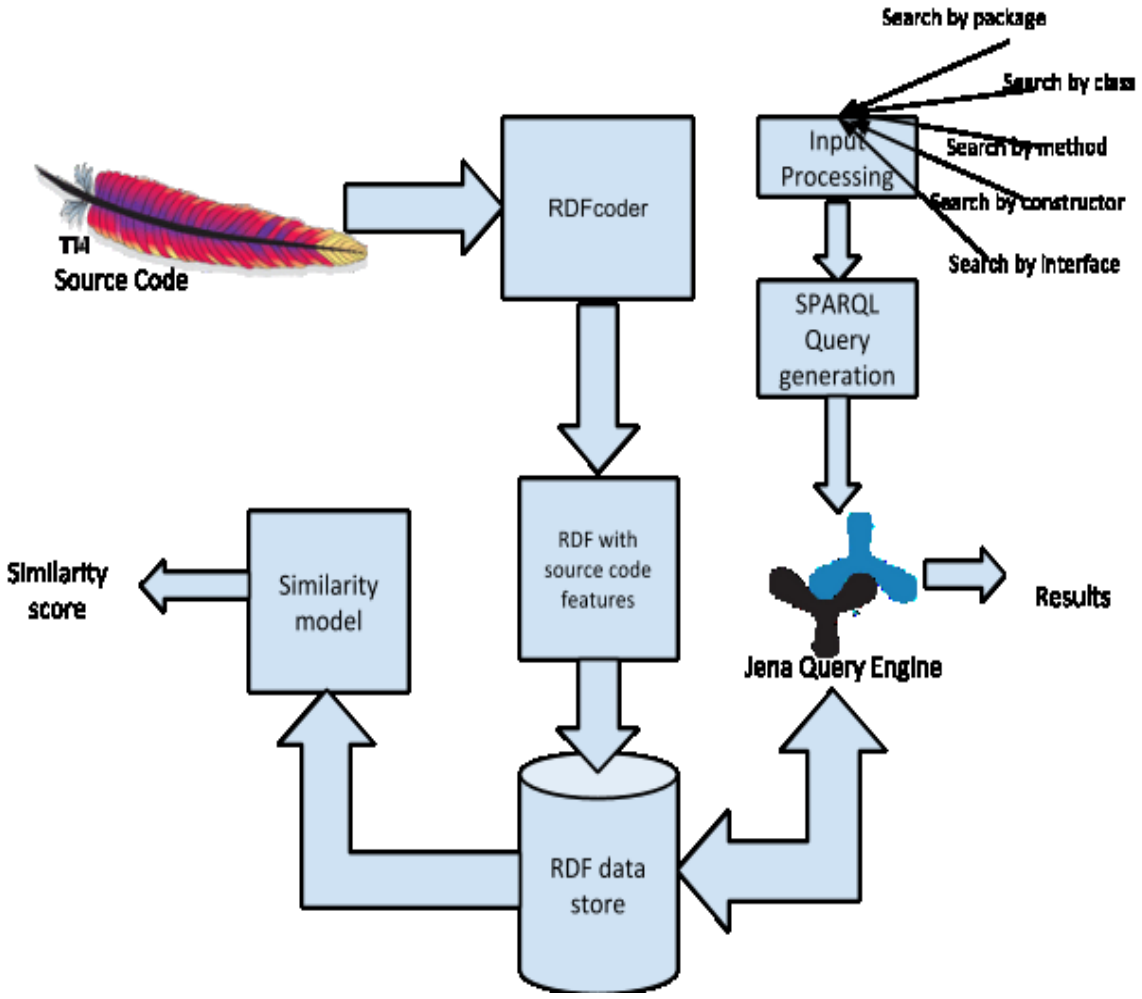

Figure 4.1 System Architecture

4.2 Source Code Crawler

We used crawler4j [26] web crawler to crawl well-known open source software repositories like Apache software Foundation [16], Github [9], Sourforge [24]. We download all the software systems available in these repositories. The download sources are stored in the file system of the web server.

4.3 Java Parser

We use RDFcoder [7] as the default parser, which parses java source code, extracts the contents, features of java source code, and generates RDF models for the downloaded code. RDFcoder uses kabbalah model as its ontology. The model represents the RDF schema used by RDFcoder to represent Java libraries information. We use RDFCoder Java API to convert Java source code to RDF model by creating a new RDF coder insatance and using JavaProfiler to generate the onology for the source code.

4.4 SPARQL Endpoint

We use Apache Jena ARQ engine [17] for executing SPARQL queries on the generated RDF models by RDFcoder for semantic searching. The Query generator generates various SPARQL queries depending on the user's input. The input is taken from Web Interface explained in section 4.5. An example SPARQL query to search the method signature and visibility of the method are shown in Table 4.1. In this example we query for signature and method with the help of the method's name (jmethod:createHeloPacket).

Table 4.1: Example SPARQL query

Example: To extract signature and visibility from the above method
select ?signature ?visibility+
where {

```
<http://www.rdfcoder.org/2007/1.0#jpackage:javachat.network.message.jmethod:createHeloPa
cket> j.0:contains_signature ?signature
<http://www.rdfcoder.org/2007/1.0#jpackage:javachat.network.message.jmethod:createHeloPa
cket> j.0:has_visibility ?visibility
}
```

4.5 Web Interface

HTML5 and jQuery are used for the frontend development of the application, seen in figure 1. The user can select search for Class, Method, Package, Constructor and Interface. Depending on the user request, the Interface changes so that the user can input the needed values for more accurate results. The user keywords much as they would provide any search engine. The Ajax call will send the appropriate keywords to the web server, which uses these keywords and creates a SPARQL query. This is executed later using Jena engine.

There are different forms for different search queries. For searching a class the user can enter the following parameters extends_class, Implements_class, contains_method or visibility, these parameters are optional for more accurate results we need more parameters. Later all these parameters are mapped to form a SPARQL query

The results are as displayed in a modal window shown in Figure 4.2.

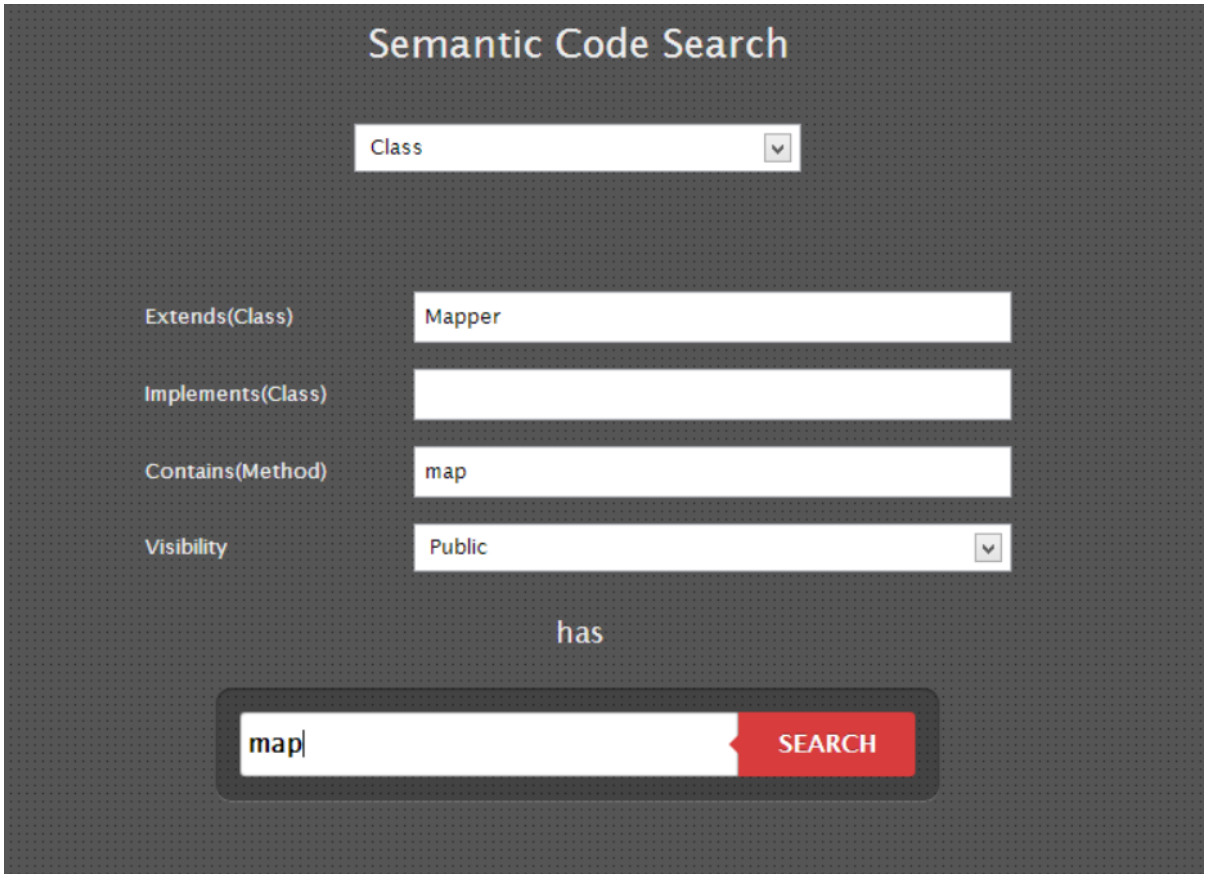

Figure 4.2 Semantic Code Search Based Web Interface

In this chapter, we have discussed technical aspects of our system. We discussed architecture and several components in our system. In the next chapter, we will discuss our system evaluation.

CHAPTER 5

RESULTS AND EVALUATION

In this chapter, we look at how the users can search the Semantic Code Search Engine and evaluate Semantic Code Search with Github. This section also includes the evaluation of Semantic similarity with the help of analyzing Apache Hadoop software system.

5.1 Experimental Setup

All the experiments run on Microsoft Azure Cloud Platform on a single node with Microsoft Windows Server 2012 R2, 3.5 GB RAM, 64-bit AMD Opteron™ processor 4171 HE X 2, and 30GB Hard Drive. JVM 1.7.0_51, GlassFish server 4.0 and Apache Jena 2.11.1 are used.

5.2 Dataset Used

We use eight different Apache Hadoop source code for evaluating semantic similarity. The extracted features from source-code are and the semantic similarity model will run on these versions. The source code analytics and the generated RDF analytics for the different versions is as shown below. We used Apache Hadoop as our dataset for evaluation as it is a very active project in Apache software foundation.

The Hadoop core source code analytics is shown in Table 5.1, which contains information like the LOC, number of comments, and number of Packages, Classes, Constructors, Methods and Fields.

Table 5.1: Hadoop Core Source Analytics

Version	0.1.0	0.5.0	0.10.0	0.15.0	0.20.0	0.22.0	1.0.0	1.1.2
LOC	13075	23597	33514	53658	95855	152459	133096	143903
Comments#	2624	3631	4891	7635	13237	19949	18006	19257
Packages#	13	21	27	34	75	109	115	119

Classes#	207	333	442	689	1174	1429	1700	1821
Constructor#	203	327	429	656	1152	1801	1552	1647
Methods#	1275	2163	3000	4696	8037	12558	11158	11880
Fields#	723	1167	1653	2521	4303	7279	6600	7190

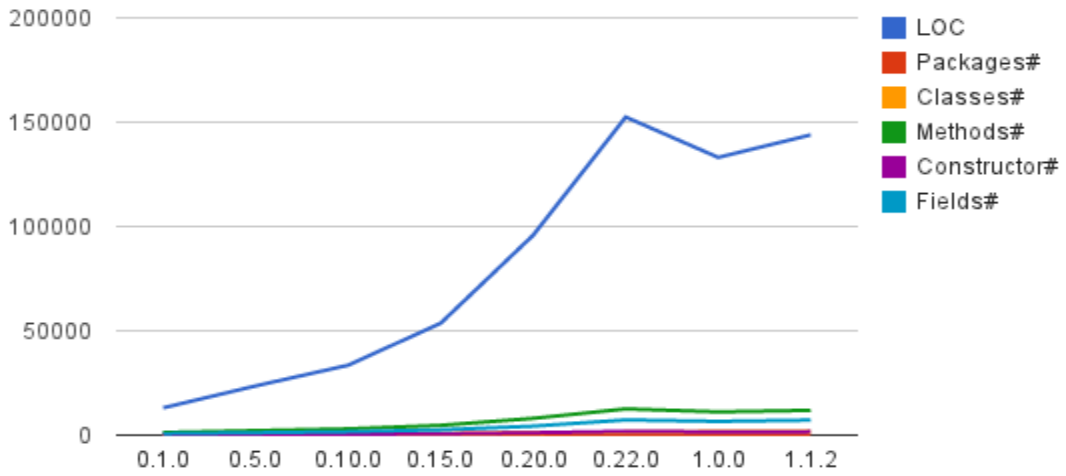

Figure 5.1 Evolution of Hadoop

Figure 5.1 gives a brief overview of the evolution of Hadoop source code repository in terms of LOC, Packages, Classes, Methods, Interfaces and Fields. It can observed from Figure 5.1 that the Apache Hadoop core codebase has been increased periodically for each version.

The Hadoop core RDF analytics is shown in Table 5.2, which contains information like the number of tuples, distinct resource URI's, predicates, subject nodes and object nodes.

Table 5.2: Hadoop Core RDF Analytics

Version	0.1.0	0.5.0	0.10.0	0.15.0	0.20.0	1.0.0	1.1.2
Triples#	22711	37717	51102	79206	137306	197177	210633
Distinct resource URI#	6020	10002	13658	21315	36783	52519	56011

Distinct Predicates#	22	22	24	24	24	24	24
Distinct Subject Nodes#	5595	9957	13524	21026	36482	52039	55507
Distinct Object nodes	6018	10000	13656	21313	36781	52517	56009

5.3 Results-Hadoop Core Similarity

We calculate similarity for each Hadoop version and the coSim similarity measurement for each version is shown in table 5.3. We can find that the coSim measurement is consistent for all the versions. The coSim measurement for the first Hadoop version (0.1.0) v/s last Hadoop version (1.1.2) is 0.2961 and the coSim measurement for the Hadoop version (1.0.0) v/s Hadoop version (1.1.2) is 0.8935.

Table 5.3: Hadoop Core coSim measurement

Version	0.1.0	0.5.0	0.10.0	0.15.0	0.20.0	0.22.0	1.0.0	1.1.2
0.1.0	1							
0.5.0	0.6711	1						
0.10.0	0.5077	0.7750	1					
0.15.0	0.3820	0.5771	0.6693	1				
0.20.0	0.3245	0.5052	0.5539	0.7515	1			
0.22.0	0.3168	0.4856	0.5368	0.7266	0.8416	1		
1.0.0	0.3079	0.4639	0.4885	0.6635	0.7694	0.8117	1	
1.1.2	0.2961	0.4572	0.2961	0.6454	0.7057	0.8148	0.8935	1

5.4 Results-Similarity at different levels

We calculate similarity between the first Hadoop version (0.1.0) and the last Hadoop version (1.1.2) and compare it with the similarity between Hadoop version(1.0.0) v Hadoop version (1.1.2). In table 5.4 we give in similarity details of Packages, Classes and Methods.

We consider similar and less similar, where similar has one as coSim measurement and less similar has >0.5 coSim measurement.

Table 5.4: Similarity between First and Latest Versions

Version	Hadoop 0.1.0 v Hadoop 1.1.2	Hadoop 1.0.0 v Hadoop 1.1.2
Score	0.2961	0.8935
Similar Packages(=1)	1	20
Less Similar Packages(>0.5)	0	81
Similar Classes(=1)	6	313
Less Similar Classes(>0.5)	12	788
Similar Methods(=1)	72	10050
Less Similar Methods(>0.5)	33	600

We calculate similarity measurement for each level in the software system. In table 15 we show the similarity of packages that contains similar classes, less similar classes and the similarity measurement for the package. We use Hadoop version (1.0.0) and Hadoop version (1.1.2) for evaluation and the package used for evaluation is "org.apache.hadoop.hdfs.server.namenode".

Table 5.5. Package Level similarity measurement

Hadoop versions	Hadoop version (1.0.0) vs Hadoop version (1.1.2)
Package Name	org.apache.hadoop.hdfs.server.namenode
Package Score	0.9087
Similar Classes#	72
Similar Classes	GetDelegationSocketServlet,EditLogOutputStream, UpgradeManagerNamenode
Less Similar Classes#	10
Less Similar Classes	JspHelper, Host2NodesMap, Result, NumberReplicas, NameNodeMXBean
Most changed class	Result (Class Score:0.73)

5.5 Results-Semantic Query Evaluation

We use Hadoop 1.1.2 software repository for our evaluation. We make the same searches on both search systems and compare the results obtained from the search for accuracy. For quicker and fair results, we use github repository search (search on the single repository Hadoop 1.1.2). As shown in Table 5.6 Semantic Query Engine performed better in terms of accuracy and structure based queries. Semnatic Query Engine was also efficient in inferring based queries like Class throwing UnknownHostException.

Table 5.6: Semantic Query Evaluation

Query	CodeSearch	Github
Class Map extends Mapper	Map Class	Listed all files which contained Mapper or Map
List all classes in package org.apache.hadoop.conf	All classes in org.apache.hadoop.conf	Listed files containing org,org.apache,org.apache.hadoop,conf
List all classes which extends Mapper	All classes which extended Mapper	Listed all files which contained Mapper
Class throwing UnknownHostException	DFShost class	Not Found
Class implements Closable	JavaSerilizatiorDeserializer class returned	Not found
Show me all the public methods in the package org.apache.hadoop.conf	listed all the methods	Listed 1139 code results which had keyword public or org.apache.hadoop.conf
List classes containing method write	Listed methods	Not Found

In this chapter, we have discussed our system evaluation. We evaluated semantic similarity on Apache Hadoop codebase and semantic query system using Github.

CHAPTER 6

CONCUSION AND FUTURE WORK

6.1 Conclusion

We developed a novel semantic code search and analytics system based on searching source code using its structure and finding the similarity between two different systems. The semantic approach for searching and similarity finding is very critical for the system. Use of Jaccard Similarity Index and Levenshtein String Distance were important as text plays a very important role in finding similarity between each statement in a RDF node. The filtering option helps in reducing the number of comparisons depending on the level the user requires.

The system when evaluated by comparing it with Github, provided more features than Github currently does. Similarity evaluation using the Hadoop Source code, the system found the similarities between different Hadoop versions.

6.2 Future Work

Semantic code search mainly focuses on structure-based search and semantic similarity, as there is a huge open source-source code available, the scalability of system is very limited with our current approach. We can use Bigdata platform for addressing this issue. For increasing the accuracy, we can work on more semantic feature extraction and custom attributes for searching. The system will perform better by ranking the obtained search results based on the structure matching with users' input.

Connecting the obtained search results with the documentation of source code would be a very great addition to the system. Also implementing the system as an Eclipse plugin would be convenient for the developers.

REFERENCES

[1] Reiss, Steven P. "Semantics-based code search." *Software Engineering, 2009. ICSE 2009. IEEE 31st International Conference on*. IEEE, 2009.

[2] Bajracharya, Sushil, et al. "Sourcerer: a search engine for open source code supporting structure-based search." *Companion to the 21st ACM SIGPLAN symposium on Object-oriented programming systems, languages, and applications*. ACM, 2006.

[3] Maletic, Jonathan I., and Andrian Marcus. "Using latent semantic analysis to identify similarities in source code to support program understanding." *Tools with Artificial Intelligence, 2000. ICTAI 2000. Proceedings. 12th IEEE International Conference on*. IEEE, 2000.

[4] Cosma, Georgina, and Mike Joy. "An approach to source-code plagiarism detection and investigation using latent semantic analysis." *Computers, IEEE Transactions on* 61.3 (2012): 379-394.

[5] Mishne, Alon, Sharon Shoham, and Eran Yahav. "Typestate-based semantic code search over partial programs." *ACM SIGPLAN Notices* 47.10 (2012): 997-1016.

[6] Thummalapenta, Suresh, and Tao Xie. "Parseweb: a programmer assistant for reusing open source code on the web." *Proceedings of the twenty-second IEEE/ACM international conference on Automated software engineering*. ACM, 2007.

[7] Rdfcoder-The java to RDF modeler (2014). The Google [Online]. Available: https://code.google.com/p/rdfcoder/.

[8] The Resource Description Framework (RDF) (2014). The W3C [Online]. Available: http://www.w3.org/RDF/.

[9] Github-Build software better, together (2014). The Github, Inc. [Online]. Available: http://github.com/.

[10] Raymond, Eric. "The cathedral and the bazaar." *Knowledge, Technology & Policy* 12.3 (1999): 23-49.

[11] Google Code (2014). The Google [Online]. Available: http://code.google.com/.

[12] Codase – Source Code Search Engine (2009). The Codase [Online]. Available: http://www.Codase.com/.

[13] Krugle – Software development productivity (2013). The Aragon Consulting Group, Inc. [Online]. Available: http://www.krugle.com/.

[14] The SPARQL Query Language for RDF (2008). The W3C [Online]. Available: http://www.w3.org/TR/rdf-sparql-query/

[15] Jaccard Index (2014). The Wikipedia [Online]. Available: http://en.wikipedia.org/wiki/Jaccard_index/.

[16] The Foundation (2012). The apache software foundation [Online]. Available: http://www.apache.org/foundation/.

[17] Apache Jena (2012). The apache software foundation [Online]. Available: https://jena.apache.org/.

[18] Levenshtein Distance (2014). The Wikipedia [Online]. Available: http://en.wikipedia.org/wiki/Levenshtein_distance/.

[19] SparsJ (2006). The SPARS Project [Online]. Available: http://demo.spars.info/.

[20] Sourceforge (2014). The Dice Holdings [Online]. Available: http://sourceforge.net/.

[21] Meanpath – The Source Code Search Engine (2013). The Meanpath [Online]. Available: http://meanpath.com/.

[22] Ohloh Code Search (2014). The Black Duck Software [Online]. Available: http://code.ohloh.net/.

[23] GitHub (2014). The Wikipedia [Online]. Available: http://en.wikipedia.org/wiki/GitHub/.

[24] SourceForge (2014). The Wikipedia [Online]. Available: http://en.wikipedia.org/wiki/SourceForge/.

[25] InstaSearch plugin [Online]. Available: http://code.google.com/a/eclipselabs.org/p/instasearch/.

[26] Crawler4j [Online]. Available: http://code.google.com/p/crawler4j/.

VITA

Yashwanth Rao Dannamaneni was born on April 16, 1991, in Andhra Pradesh, India. He completed his Bachelor's degree in Information Technology from Amrita Vishwa Vidhyapeetham, Bangalore, India in 2012. He then pursued his Masters in Computer Science at the University of Missouri-Kansas City (UMKC). Upon completion of his Master's program, Mr.Yashwanth Rao Dannamaneni will be joining Sprint as a Data Science Intern.