An open framework for semantic code queries on heterogeneous repositories

Conference or Workshop Item

An Open Framework for Semantic Code Queries on Heterogeneous Repositories

Tian Zhang[†‡], Minxue Pan[†§*], Jizhou Zhao[†‡], Yijun Yu[¶] and Xuandong Li[†‡]
[†]State Key Laboratory for Novel Software Technology, Nanjing University
[‡]Department of Computer Science and Technology, Nanjing University
[§]Software Institute, Nanjing University, P. R. China
[¶]Department of Computing & Communications Centre for Research in Computing, The Open University, UK
Email:{ztluck,mxp}@nju.edu.cn, zjz@seg.nju.edu.cn, y.yu@open.ac.uk, lxd@nju.edu.cn

***Abstract*—To help developers understand and reuse programs, semantic queries on the source code itself is attractive. Although programs in heterogeneous languages are being controlled for collaborative software development, most queries supported by various source code repositories are based either on the metadata of the repositories, or on indexed identifiers and method signatures. Few provide full support to search for structures that are common across different programming languages and different viewpoints (hence heterogeneous). To facilitate understanding and reuses, in this paper, we propose a novel source code query framework that (1) transforms source code to a unified abstract syntax format, and handles heterogeneity (non-isomorphism) at the abstract syntax level; (2) stores source code on a cloud-based NoSQL storage in MongoDB; (3) rewrites semantic query patterns into the NoSQL form. The efficiency of the framework has been evaluated to support several open-source hosting platforms.**

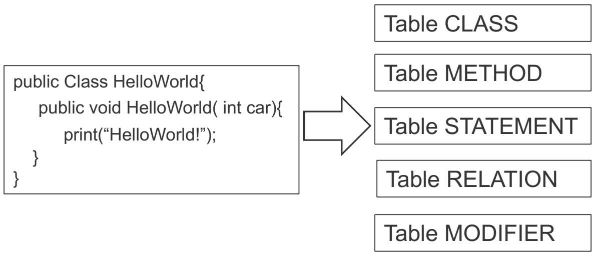
I. Introduction

In the past decade, code-hosting repositories are growing prosperously, due to the explosively increasing number of open-source projects. Representatives such as GitHub [1], SourceForge [2], Google Code [3], Sourcerer [4] have become quite a popularity among software developers for their capability to supply various codes for references. Available existing codes that one can acquire are more than ever before. However, code reuse and analysis is still seldom seen in practice. The main reason is code in need is difficult to find. The state of the art code repositories like GitHub, SourceForge and Krugle [5] provide only keyword search function which limits the effectiveness of finding appropriate codes for a specific application [6]. Often, keyword matching does not mean any resemblance in the software functions. For example, an “iterator” statement in Java can be used to do a element search in one program, or a quick sort in another. Even if one is lucky enough to find a piece of code by keyword matching that meets the requirements, the code may be too complex to be understood and therefore cannot be adapted to the new program. It is widely acknowledged that semantic code query is the solution, and many studies have been conducted on the subject. In [7], relations between expressions, units and modules can be specified with the specification language of the CARE system. In [8], a contact based specification is used to query the relations between methods and components. In [9], a semantic indexing based specification is adopted in the query for the component relations. Some code repositories also support simple semantic queries, such as Codifier [10]. Unfortunately, none of these work support the semantic query of the relations of classes and objects in the object-oriented (OO) programs. Nowadays, most modern languages have embraced the OO concepts, which results in that the program designs are OO based. For example, many programs have used design patterns which essentially are OO based to fulfill the intended functions. As a consequence, some preliminary work have been conducted to address the problem of querying relations among OO constructs. In [11], a query language JTL is proposed which is Datalog alike and can query element relations in Java programs. In [12], a OO query language called .QL which queries for program structure is introduced. .QL is based on SemmleCode, which is within Eclipse to query Java programs. To our best knowledge, all these pieces of work focus on some specific programming languages. If one needs to perform queries on more than one languages, it is required that different querying language be learned, which is quite a burden for developers, as source code query is meant to ease the coding process by reusing similar code snippets. How to do code query on various programming languages at a semantic level becomes a problem badly in need of solution.

When semantic code query is supported, the choice of the storage format for the programs becomes an interesting problem. Most repositories use relational database, such as MySQL [4], [5] or MSSQL [10]. For syntactic code queries like keyword matching, relational database is acceptable for its performance. However, when it comes to semantic code queries concerning OO relations, relational database can be inefficient. Usually, OO relations are about multiple classes and objects. Therefore, to extract one snippet, multiple code snippets need to be scanned, which can cause problems for relational databases. The first problem is the high cost of code flattening in the storing process. For storing in relational databases, codes need be decomposed into different information parts including structure information, index information, text information, etc.. For example, for the code snippet in Figure 1, information has to be scattered into five tables. The second problem is that when the target source code is found and to be presented, they need to be reconstructed from flattened codes. For example, if one wants to find a print statement used in the public class “HelloWorld”, all the five tables will be joined to produce the result. The third problem is that the performance of the relational databases could be severely affected when there are massive codes to be stored. These three problems restrict the performance of code query

[*] Corresponding author

in relational databases.



```
public Class HelloWorld{
    public void HelloWorld( int car){
        print("HelloWorld!");
    }
}
```

Fig. 1. Code decomposition for storing in relational database

To address the aforementioned problems, in this paper we propose an open framework which facilitates the semantic code query on heterogeneous repositories. We propose to a new query language JIns^{+} which is the the extension of a declarative code instrument language JIns [13] to perform code queries on heterogeneous OO programs. The JIns language is designed for context- sensitive instrumentation tasks of Java programs. We chose it as the basics of our query language JIns^{+} based on the following observations:

- JIns supports the query of OO relations in Java programs;
- JIns' quantified predicate logic expressions provides expressive query rules;
- JIns' SQL-like style is easy to learn and comprehend by most programmers.

JIns^{+} adopts the JIns basic syntax, but is not restricted to query Java programs. It has the ability to query on various OO programs including C++, C#, Java, Scalar, etc., which is achieved by not associating any specific languages with its syntactic elements. Instead, a semantics based on the general OO concepts is given. When querying on a particular language, the general semantics is mapped to the language specific syntax, which will be used for the code match that finds the codes satisfying the queries. With this two- layered mapping, JIns^{+} can query on heterogeneous OO programs. For example, to find a class *cname*'s direct subclass in Java, one can write in JIns^{+} as:

```
find all c:class satisfying exist c′:class
where c′.name=cname && c extend c′
```

The statement `c extend c′` is not associated with the Java code, but given a semantics adjusted from the UML specification whose purpose is to provide a unified modeling approach for software systems with OO concepts despite the language differences. Informally, c extend c' where c and c' are both classes means that c' is a generalization of c. This semantics is used to make mappings to each specific programming language that the repository supports. For C++, this generalization between c and c' is mapped to the syntax of c:c'. So when we want to search for a code snippet satisfying the query statement in a C++ program, there is no need to learn a new query language.

For storing codes, we propose to use NoSQL databases. Compared with traditional relational databases which require that schemata be defined as relational tables, NoSQL databases are built to allow the insertion of data of flexible structures [14], which is suitable for storing codes since their structures vary across different programs. Among miscellaneous NoSQL databases, MongoDB [15] is an open-source document database, and the leading NoSQL database. We chose MangoDB as the our database, for it adopts a document-oriented storage approach. Usually, OO source codes have more complex structure and deeper hierarchies than procedural codes. MangoDB's nested document storage is able to reserve most of the nested structure of OO codes, which makes it suitable for the task. Another advantage of MongoDB is its scalability that it can cope with big data very well. There are vast open-source projects nowadays. As a matter of fact, we have already collected more than ten thousand of projects whose occupied storage has reached to the scale of terabyte, so the choice of NoSQL database would be very reasonable.

MongoDB stores data in the format of JSON [16], which is a lightweight data-interchange format. JSON is easy for humans to read and write, and for machines to parse and generate. It is based on a subset of the JavaScript Programming Language, whose grammar is close to most OO languages. Nevertheless, we still need to transform the source codes to the JSON format. One way to do this is to write transforming rules that directly transform the source codes to corresponding JSON data. Since there is no standard transformation out there, this method is ad-hoc, and have two major problems. One is that we have to come up with a unified schema for the JSON data to be stored in MangoDB. Otherwise it would be impossible for the JIns^{+} queries to understand the data. Even though the schema has been well defined, we still need to write a transformation procedure for each OO language respectively according the transformation rules, as each OO language has different syntax. This can be extremely tedious and error-prone. Therefore, we employ the other approach, which transforms the codes to an standard intermediate format, and then to the JSON data. We use FreeTXL, an implement of TXL [17], as the intermediate format specifying language. TXL is a programming language specifically designed to support computer software analysis and source transformation tasks, and is the evolving result of more than fifteen years of concentrated research on rule-based structural transformation [18]. It specifies the program structures to be transformed and the transforming rules, which can be applied to all kinds of programming languages. Besides, in FreeTXL, transformation rules for 18 mainstream programming languages such as Java, c++, c# and Python have already been predefined and implemented as runnable programs. The results after the transformation are a set of XML files which can be easily transformed to JSON data via JDOM [19]. With this two-step transforming, we have avoided the work of writing transformation rules and the target code schemata, while at the time gain the prize that the results of transforming heterogeneous programming languages are in accordance with a unified schema.

By combining the query and storage techniques, we propose a framework for semantic code queries on heterogeneous repositories. The framework is open, in the sense that programs written in new languages can be easily added to the repository, and can be searched without learning new query languages. To summarize, we make two major contributions in this paper:

- We proposed a semantic code query language JIns^{+}. With a two-layered mapping, JIns^{+} hides the heterogeneity of different OO programming languages and can query the OO relations across various OO languages;

- We proposed a NoSQL-based source storage approach. With a two-step transforming, heterogeneous OO source codes can be stored in the MangoDB database and can reach a scale of practical use.

The rest of this paper is organized as follows. Section 2 briefs the framework. Section 3 presents the details of the storage approach with MangoDB. Section 4 presents the query language , with which we perform the semantic code queries on the repositories. Section 5 shows two case studies. Section 6 discusses related work from recent years. Finally, Section 7 concludes this paper.

II. Framework Overview

In this section, we will briefly introduce our proposed framework, which is illustrated in Figure 2. The framework consists two components: the code storage component and the code query component.

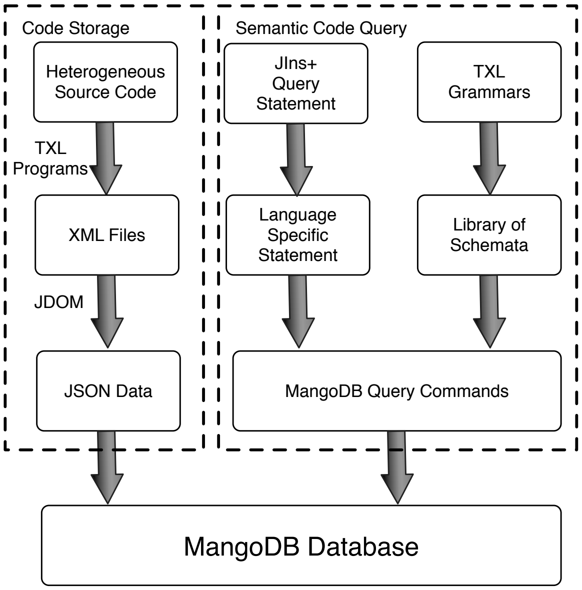


Fig. 2. The overview of the framework

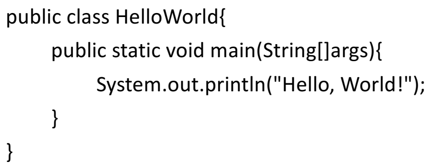
The code storage component transforms source codes of heterogeneous languages to the data that can be stored in the MangoDB database. As it is mentioned in introduction, we adopt a two-step transformation strategy. First, the source codes that acquired from online repositories are transformed into a group of XML files by applying TXL programs. Then the XML files are transformed with JDOM into JSON data, which can be accepted by the MangoDB database.

The semantic code query component performs the JIns$^+$ query commands on the code repositories. The query is not directly executed on the MangoDB query engine, but through a two-layered mapping. First the syntax of the JIns$^+$ query statements are mapped to the language specific syntax. At the same time, a schema of the nested structure of the JSON data is derived by analyzing the TXL grammar. Then by combining the two, a MangoDB query command is generated and executed on the database, which produces the query results. The technical details of the two components are presented in the next two sections.

III. Code Storage with MangoDB

In this section, we will introduce how to store heterogeneous source codes in MangoDB. The source codes we stored were acquired from the online code repositories such as GitHub, Bitbucket, or Google Code, etc.. We used a web crawler to automatic download open source codes from different repositories. The codes are written in different programming languages, however, with our two-step transforming, the heterogeneity of different codes are well handled. First, we use FreeTXL to transfer different kinds of source coded into XML files. It helps a lot that transformation grammars of main programming languages are offered officially. In the following, we will show the transforming and storing process with a Java code snippet example.

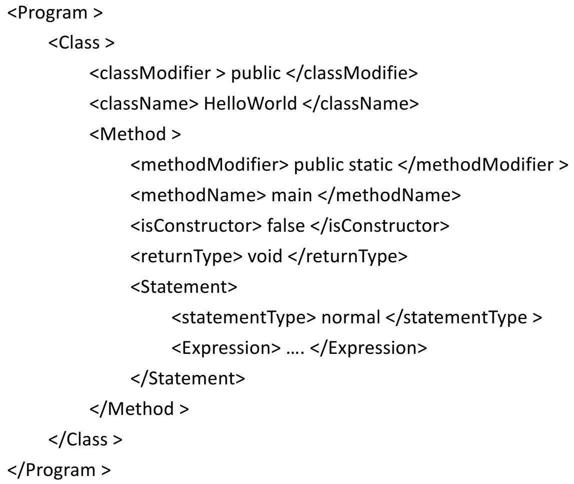
To make the illustration concise, we use the well-known "HelloWorld" snippet as the example:



```
public class HelloWorld{
    public static void main(String[]args){
        System.out.println("Hello, World!");
    }
}
```

Fig. 3. The "HelloWorld" Java code snippet

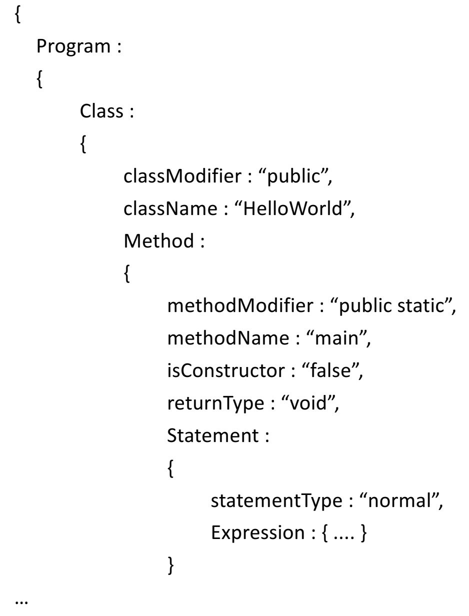
We use FreeTXL to transfer source code into XML using the JAVA grammar offered by the TXL group, the resulting XML file is shown in Figure 4.



```
<Program >
    <Class >
        <classModifier > public </classModifie>
        <className> HelloWorld </className>
        <Method >
            <methodModifier> public static </methodModifier >
            <methodName> main </methodName>
            <isConstructor> false </isConstructor>
            <returnType> void </returnType>
            <Statement>
                <statementType> normal </statementType >
                <Expression> …. </Expression>
            </Statement>
        </Method >
    </Class >
</Program >
```

Fig. 4. The XML file of the "HelloWorld" code snippet

To clarify, the XML tree shown in Figure 4 has been trimmed to make it easier to describe its main structure. The original XML file generated by FreeTXL has more details. In this XML tree, the root node is "Program". The node "Class" is the only child of "Program", which means this program file only has one class. “classModifer”, “className” and “Method” are the three children of “Class”. The first two nodes contain the configuration information of the class while the last node leads to the method inside of the class. “Method” has five children. “methodModifier”, “methodName”, “isConstructor” and “returnType” show the configuration information of the method. “Statement” includes the information of statements inside the method. The “statementType” node is one of the children of “Statement”. There are sixteen different types of statement like “for_statement” or “if_statement”, but in this method, the statement “System.out.println(“Hello, World!”);” is just a ordinary expression. The “expression” node describes the detailed information of the expression statement which is omitted here. Then we use JDOM to transfer XML into JSON, and the result is shown in Figure 5. Compared with the XML file in Figure 4, we can tell that XML nodes are corresponding to JSONs structural layers, which guarantees that the two-step transformation can be done straightforward and seamlessly.



```
{
   Program :
   {
         Class :
         {
               classModifier : “public”,
               className : “HelloWorld”,
               Method :
               {
                     methodModifier : “public static”,
                     methodName : “main”,
                     isConstructor : “false”,
                     returnType : “void”,
                     Statement :
                     {
                           statementType : “normal”,
                           Expression : { .... }
                     }
…
```

Fig. 5. The JSON data of the “HelloWorld” XML file

JSON is the storage format of MongoDB, so we can use MongoDB storing driver to add the JSON data into the database. Each Java source file corresponds one JSON data, and in this way we can store a Java project. For projects written in other programing languages, we can employ the two-step transformation in a similar way and store heterogeneous codes in the same MangoDB repository.

IV. Semantic Code Query

A. Query Rules

JIns$^+$ defines its query rules in a declarative approach. During the query, the user needs to specify the type of the query target and the conditions that need to be satisfied by the target and other elements related to the target. Currently, our work intends to construct a source query platform only for code written in OO programming languages such as Java, C #, Smalltalk, which have three features: encapsulation, inheritance and polymorphism.

Based on the query rules of JIns, we custom a set of query rules for JIns$^+$:

$$\begin{aligned}
&S \Rightarrow find\ \{all\}\{exist\}\ Id : T\ satisfying\ CS\\
&T \Rightarrow variable\\
&\qquad\quad |\ statement\\
&\qquad\quad |\ method\\
&\qquad\quad |\ class\\
&\qquad\quad |\ interface\\
&CS \Rightarrow \{exist\ Id : T\}\{all\ Id : T\}\ where\ CE\\
&CE \Rightarrow\ CE\ \&\&\ CE\\
&\qquad\quad |\ CE\ ||\ CE\\
&\qquad\quad |\ !\ CE\\
&\qquad\quad |\ (\ CE\ )\\
&\qquad\quad |\ Id.ATT = \ 'value'\\
&\qquad\quad |\ Id\ REL\ Id\\
&ATT \Rightarrow name\\
&\qquad\quad |\ dataType\\
&\qquad\quad |\ specialType\\
&\qquad\quad |\ returnType\\
&\qquad\quad |\ paramsType\\
&REL \Rightarrow extend\\
&\qquad\quad |\ use\\
&\qquad\quad |\ change\\
&\qquad\quad |\ isIn\\
&\qquad\quad |\ call
\end{aligned}$$

In the above rules, the term Id is the symbolic name represents the target that one wants to query about. The target can be associated with quantifiers including *all* and *existing*. Types of the targets available now are `Class, Interface, Method, Statement` and `Variable`. There are two kinds of query conditions that JIns$^+$ supports: the attribute query (ATT) and the relation query (REL). Multiple query conditions can be combined with conjunctions and disjunctions.

For each target (class, method, etc.), we define its properties with attribute values, which is in accordance with JIns [13]. Here we just give an example of the target variable. For more information, the reader can refer to [13]. Target variable has three properties: name, dataType and specificType The name represents the identifier of a variable in a program. The dataType represents the type of a variable, for example, if a variable is declared as a stack. The specificType represents the scope of a variable. The value “field” of specificType means that the variable is a field of a class, the value “local” means that the variable is a local variable of the program, and the value “parameter” means that the variable is a parameter of a method. Note that most of these attributes exist in almost all the programming languages, and therefore can be mapped to each language without much effort.

However, OO relations in different languages can vary enormously. For example, the generalization relation in C++ can be written roughly as $subclass\ :\ superclass$, while in

Java it is $subclass\ extends\ superclass$. We have surveyed multiple OO languages and summarized five relations. The relations are not associated with any specific language syntax. Instead, they are given a semantics independent of programing languages, which later can be mapped to language specific elements. The relation REL in the term $Id_1\ REL\ Id_2$, can be one of the followings:

- $extend$ stands for the generalization relation, which means Id_2 is the generalization of Id_1.
- use reflects the relationship that if Id_1 uses Id_2 to fulfill its function. There are three use relations: (1) statement element use variable element; (2) method element use variable element; and (3) class element use variable element.
- $isIn$ relation specifies that Id_1 is declared in Id_2. It has some similarity to use, while use does not require one element is declared in the other. There are four $isIn$ relations: (1) statement element isIn statement element; (2) statement element isIn method element; (3) method element, statement element or variable element isIn class element; and (4) method element isIn interface element.
- $change$ relation represents that Id_1 changes the value of Id_2, where Id_2 must be a variable and Id_1 can be statement, method or class.
- $call$ means a method Id_2 is called by a statement or another method Id_1.

Now we can write code query commands following the rules of JIns$^+$. For example, if we want to query a class named ‘HelloWorld’, the query statement should be like:

```
find c:class satisfying where c.name ==
``HelloWorld''
```

The attributes and relations in the query statement is mapped to the language specific syntax in the two-layered mapping style, and parsed to commands that MangoDB understands, which is explained in the following section.

B. Parse of Query Statements

In our work, source code in MongoDB is stored in a nested structure. We can see from Figure 4 that Program has Class nested, Class has Function nested, Function has Statement nested and so on. If we want to search for a statement with a property of ‘Normal’, we can write a query statement like:

```
find s:statement satisfying
s.statementType='Normal'
```

After parsed to MongoDB query, it should be like:

```
Find('program.class.function.statements.
statementType': 'Normal');
```

Apparently this MongoDB query can be split into three parts:

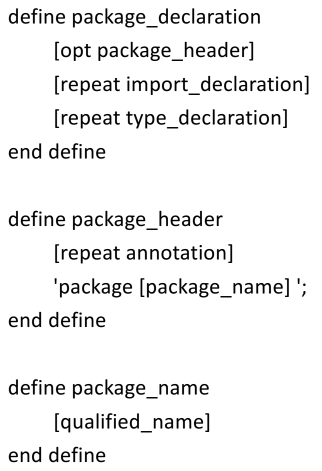
- The query target, which is ‘statementType’.
- The nested structure of the query target, which is: ‘program.class.function.statements.’.
- The matching condition of the query, which is ‘Normal’.

Back to our work, to search in MongoDB, first the query target and the matching condition have to be extracted from the query statement. Second, we need to reach out for the nested structure of the query target. Third, we compose the query target, the matching condition and the nested structure together to get the valid MongoDB Find query. Among the three parts, the query target and the matching condition can be easily extracted, which we will not discuss further.

In order to obtain the nested structure of the query target, we need to acquire the original place that defines its nested structure. As the keys in JSON corresponds to the labels in XML, the two formats are highly similar to each other, which means the nested structure of JSON data can be extracted from XML schema. The nested structure of XML files is generated by FreeTXL according to the TXL transformation grammars. Therefore, the TXL grammars are the origin.

TXL grammar is a set of rules to parse source code into XML. Elements of source code are mapped into nodes of a XML tree. In other words, TXL grammar contains the information of all possible nested structure of source code. Analyze TXL grammars offered officially, we can build a library of schemata which includes all possible nested structure of every semantic element. Because the query target must be a code element in our work, we can always get its nested structure from the library. Basically, every programming language we support has a corresponding schema in the library of nested structure. For illustration, here we only take the Java language as an example.

Figure 6 shows the structure of the Java TXL grammar. Here, we will focus more on the organization of the grammar rather than how the grammar works in the transformation.



```
define package_declaration
    [opt package_header]
    [repeat import_declaration]
    [repeat type_declaration]
end define

define package_header
    [repeat annotation]
    'package [package_name] ';
end define

define package_name
    [qualified_name]
end define
```

Fig. 6. Structure of Java TXL grammar

We can see in Figure 6 that the grammar itself is nested. The first layer is ‘package_declaration’, which means the declaration of packages of Java code. The second layer named ‘package_name’ is contained by the first layer. Likewise, the third layer named ‘qualified_name’ is contained by the second layer. The grammar is organized in this way and the last layer does not contain anything. The nested structure of the grammar is corresponding to the nested structure of the XML file transformed, which is exactly the nested structure of the corresponding JSON file.

To generate the nested structure more conveniently, we can abstract the grammar into a digraph. Every 'define' part of the grammar can be extracted as a node of code element. Edges between nodes mean the include-relationship of them. In this way, the attempt of nested structure of a code element becomes the attempt of the path from the root node of the digraph to the relative node of code element.

For a single target node, basically all its paths started from the root node have to be generated to ensure the completeness of the query. However, the digraph we constructed contains lots of rings, which makes it impossible to achieve all full paths. Therefore, we adopt a compromising strategy to collect only the paths appearing more frequently as follows:

- Step 1, pick up a node as the target and traverse through the digraph in a depth-first way to get all the simple paths from the root node to the target node, as well as the rings along the simple paths. If all the nodes have been handled, exit.
- Step 2, pick out a simple path and combine it with the possible rings. The max repeat times of a ring can be assigned by the user. Multiple paths may be generated from a simple path. Step 3, repeat Step 2 until there is no simple path left. Go back to Step 1.

All paths generated by the strategy constitute the library of possible nested structure.

Now we can put the query target, one of the possible nested structure and the matching condition together, and get one valid MongoDB query. This query is called valid, only means that it can be directly run on MongoDB. It is not guaranteed that this single query can always return the result. In order to ensure the return of result, we may have to construct one query for every possible nested structure and run every query on MongoDB.

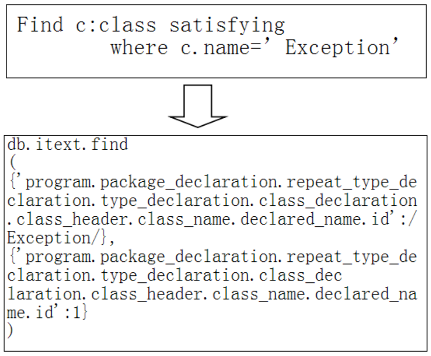
C. Return Results

MongoDB presents results in JSON, ideally we can do transformation to get the original source codes. However, this is neither possible, nor efficient. So in this article, we store a copy of source code in the form of strings instead. We get the lines of source code according to the result from MongoDB, and extract the corresponding source code in the form of strings to return.

V. Case Study

In this part of the article, we will use a simple case to show the feasibility of our approach. We obtain the source code of an open source software named IText from SourceForge. 24 classes of IText are picked out randomly for experiments. Using FreeTXL, we transfer them into XML files, and JDOM to JSON data. Then we use the JAVA driver of MongoDB to store them into a collection named 'IText'.

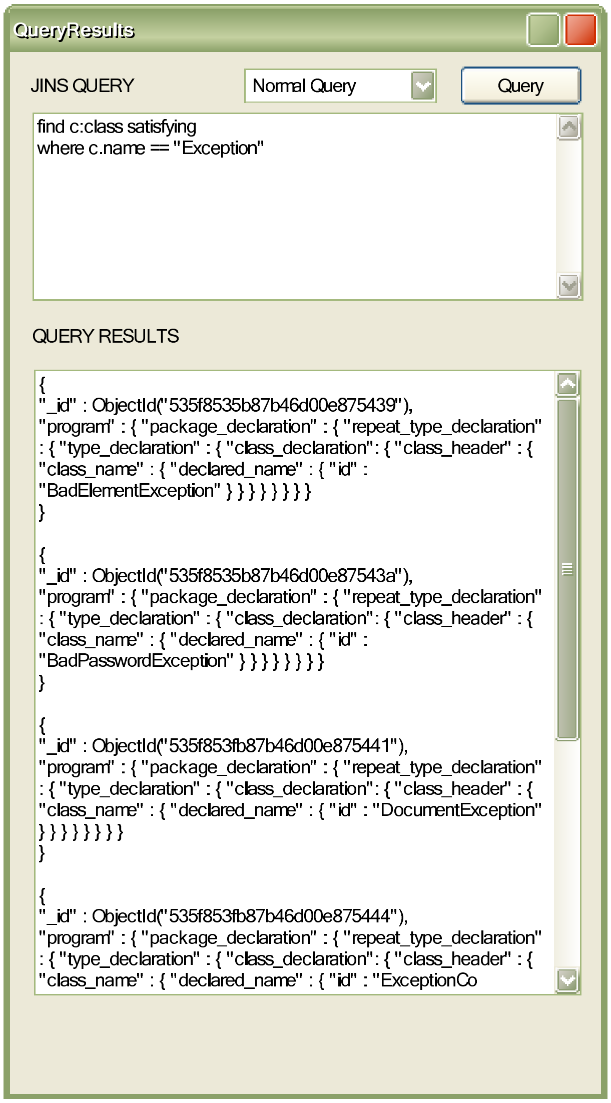
If we want to understand the exception mechanism of IText, the first idea may be to find all the classes which have a name containing the key word 'Exception'. We write the query statement and parse the query statement to MongoDB queries. The query statement and one of the possible MongoDB queries are showed in Figure 7.



```
Find c:class satisfying
        where c.name=' Exception'
```

```
db.itext.find
(
{'program.package_declaration.repeat_type_de
claration.type_declaration.class_declaration
.class_header.class_name.declared_name.id':/
Exception/},
{'program.package_declaration.repeat_type_de
claration.type_declaration.class_dec
laration.class_header.class_name.declared_na
me.id':1}
)
```

Fig. 7. Class Name Query



```
find c:class satisfying
where c.name == "Exception"
```

```
{
"_id" : ObjectId("535f8535b87b46d00e875439"),
"program" : { "package_declaration" : { "repeat_type_declaration"
: { "type_declaration" : { "class_declaration": { "class_header" : {
"class_name" : { "declared_name" : { "id" :
"BadElementException" } } } } } } } }
}

{
"_id" : ObjectId("535f8535b87b46d00e87543a"),
"program" : { "package_declaration" : { "repeat_type_declaration"
: { "type_declaration" : { "class_declaration": { "class_header" : {
"class_name" : { "declared_name" : { "id" :
"BadPasswordException" } } } } } } } }
}

{
"_id" : ObjectId("535f853fb87b46d00e875441"),
"program" : { "package_declaration" : { "repeat_type_declaration"
: { "type_declaration" : { "class_declaration": { "class_header" : {
"class_name" : { "declared_name" : { "id" : "DocumentException"
} } } } } } } }
}

{
"_id" : ObjectId("535f853fb87b46d00e875444"),
"program" : { "package_declaration" : { "repeat_type_declaration"
: { "type_declaration" : { "class_declaration": { "class_header" : {
"class_name" : { "declared_name" : { "id" : "ExceptionCo
```

Fig. 8. Partial Results of Class Name Query

By executing the MongoDB query, we can get the result, which is shown in Figure 8. We didn’t include the actual source codes corresponding to the query results here, since they have the same meaning as the query results and are only different in the presentation forms.

Then we conducted another query experiment. We know that classes dealing with exception are usually extended from Java Exception classes. If we want to find classes which extend the IOException class, we can write a query statement, as shown in Figure 9, along with one of the corresponding MongoDB queries.

```
find c1 : class satisfying
        exist c2 : class
        where c1 extend c2 && c2.name= ’IOException’
```

```
db.itext.find
(
{’program.package_declaration.repeat_type_declaration
.type_declaration.class_declaration.class_header.opt_
extends_clause.extends_clause.list_type_name.type_nam
e.qualified_name.reference.id’:’IOException’},
{’program.package_declaration.repeat_type_declaration
.type_declaration.class_declaration.class_header.opt_
extends_clause.extends_clause.list_type_name.type_nam
e.qualified_name.reference.id’:1}
)
```

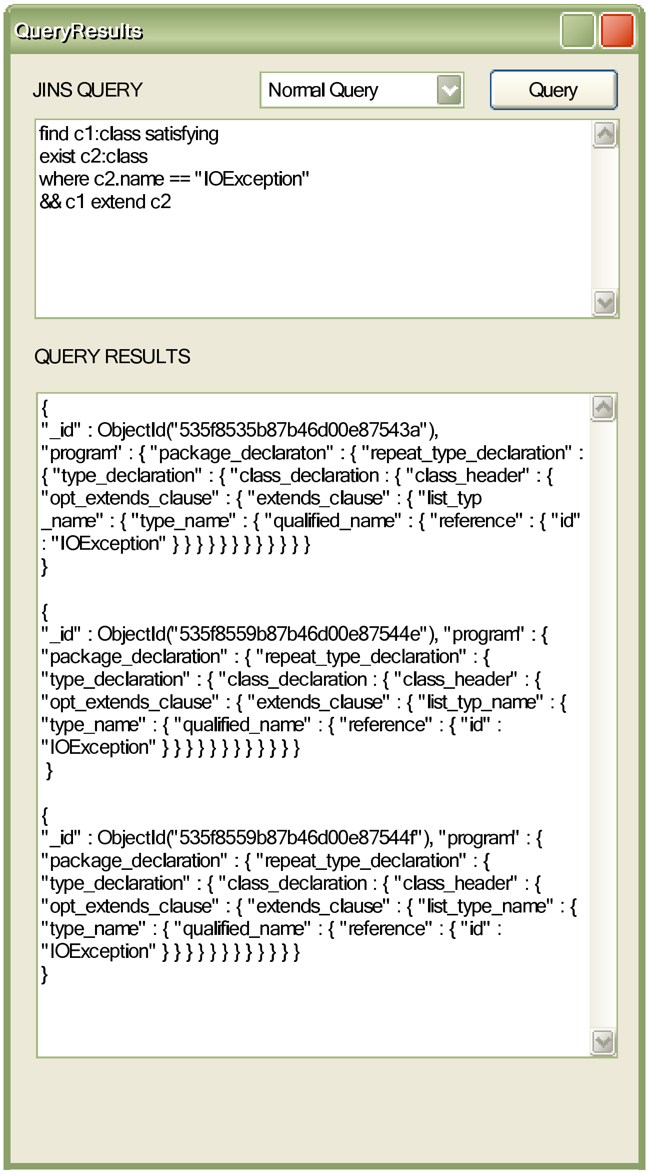
Fig. 9. Class Extend Query

We run it in MongoDB and get the output shown in Figure 10. We only return the name of classes because the return code will take too much space.

VI. Related Work

Code query plays an important role in software analysis and reuse. Applications can be found in reverse engineering, software comprehension, coding conventions checking, and so on. Besides the work mentioned in the introduction, there are many other studies concerning the problem of code query language design. One of the most expressive code query language is Prolog [20], which is logic based and Turing complete. JTransformer [21] is also logic based but less expressive. Other languages such as Grok [22], JRelCal [23] and Rscript [24] are based on relational calculus, while .QL [12], SemmleCode [25] and JGraLab [26] are relational algebra based. Among them, .QL and SemmleCode are the only two languages that explicitly support the notion of object-oriented programming. However, like many other languages such as .QL and JGraLab, they are designed only for a specific language which is Java in this case.

Code storage in database is the most efficient way to handle large quantity of available source codes. Although the NoSQL databases are gaining more attention, besides some exception such as BARISTA [27] which is an Eclipse plugin that can only query code inside Eclipse project, most existing repositories still use relational databases to store code information. Sourcerer uses three models to extract information from source code and MySQL to store the information. CodeQuest [28] uses MS SQL server or DB2 to store its code information. SemmleCode is based on H2 or MS SQL server. Recently, there are a few work turning to NoSQL databases to store codes. Work [29] stores codes in the graph form to the database Neo4J [30]. The goal of the work is not to query codes but model and discover vulnerability code patterns, so the codes were transformed into abstract syntax tree (AST) and stored. Work [31] also uses Neo4J, and tries to store the AST data together overlaid with other graph such as control flow graph. This work is still in process, so the storing paradigm is ad-hoc. Even though the paradigm was made mature like work [29], since there are very few routines for languages that already exist to transform codes to AST, most of the transforming routines would have to be developed to handle various heterogeneous programming languages. Compared with these pieces of work, our approach adopts the document based database MangoDB which can utilize abundant XML transformation techniques.



```
find c1:class satisfying
exist c2:class
where c2.name == "IOException"
&& c1 extend c2
```

```
{
"_id" : ObjectId("535f8535b87b46d00e87543a"),
"program" : { "package_declaraton" : { "repeat_type_declaration" :
{ "type_declaration" : { "class_declaration : { "class_header" : {
"opt_extends_clause" : { "extends_clause" : { "list_typ
_name" : { "type_name" : { "qualified_name" : { "reference" : { "id"
: "IOException" } } } } } } } } } } } }
}

{
"_id" : ObjectId("535f8559b87b46d00e87544e"), "program" : {
"package_declaration" : { "repeat_type_declaration" : {
"type_declaration" : { "class_declaration : { "class_header" : {
"opt_extends_clause" : { "extends_clause" : { "list_typ_name" : {
"type_name" : { "qualified_name" : { "reference" : { "id" :
"IOException" } } } } } } } } } } } }
 }

{
"_id" : ObjectId("535f8559b87b46d00e87544f"), "program" : {
"package_declaration" : { "repeat_type_declaration" : {
"type_declaration" : { "class_declaration : { "class_header" : {
"opt_extends_clause" : { "extends_clause" : { "list_type_name" : {
"type_name" : { "qualified_name" : { "reference" : { "id" :
"IOException" } } } } } } } } } } } }
}
```

Fig. 10. Partial Result of Class Inheritance Query

VII. Conclusion

In this article, we propose a framework to help open source users better query and reuse source code in different heterogeneous repositories. We propose the idea to built a query platform, transform source in heterogeneous repositories to JSON data and use MongoDB to store them. We propose a set of query rules based on JIns query rules and provide the method to parse query statements to MongoDB query commands.

The contribution of our work is that it bridges heterogeneous repositories together, makes up for the lack of semantic query of open source hosting platforms, which results in making it easier for users to query and reuse open source code. Based on the NoSQL database MongoDB, our platform performs well when the scale of source code grows to the big data level.

REFERENCES

[1] GitHub, https://github.com/, 2015, [Online; accessed 20-March-2015].

[2] SourceForge, http://sourceforge.net/, 2015, [Online; accessed 20-March-2015].

[3] Google Code, https://code.google.com/, 2015, [Online; accessed 20-March-2015].

[4] S. Bajracharya, T. Ngo, E. Linstead, Y. Dou, P. Rigor, P. Baldi, and C. Lopes, "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*, ser. OOPSLA '06. New York, NY, USA: ACM, 2006, pp. 681–682.

[5] K. Krugler, "Krugle code search architecture," in *Finding Source Code on the Web for Remix and Reuse*, 2013, pp. 103–120.

[6] S. P. Reiss, "Semantics-based code search," in *Proceedings of the 31st International Conference on Software Engineering*, ser. ICSE '09. Washington, DC, USA: IEEE Computer Society, 2009, pp. 243–253.

[7] D. Hemer and P. Lindsay, "Supporting component-based reuse in care," *Aust. Comput. Sci. Commun.*, vol. 24, no. 1, pp. 95–104, Jan. 2002.

[8] J.-J. Jeng and B. H. C. Cheng, "Specification matching for software reuse: A foundation," *SIGSOFT Softw. Eng. Notes*, vol. 20, no. SI, pp. 97–105, Aug. 1995.

[9] A. Mili, R. Mili, and R. Mittermeir, "Storing and retrieving software components: A refinement based system," in *Proceedings of the 16th International Conference on Software Engineering*, ser. ICSE '94. Los Alamitos, CA, USA: IEEE Computer Society Press, 1994, pp. 91–100.

[10] A. Begel, "Codifier: a programmer-centric search user interface," in *Workshop on Human-Coputer Interaction and Information Retrieval*, October 2007.

[11] T. Cohen, J. Y. Gil, and I. Maman, "Jtl: The java tools language," *SIGPLAN Not.*, vol. 41, no. 10, pp. 89–108, Oct. 2006.

[12] O. Moor, D. Sereni, M. Verbaere, E. Hajiyev, P. Avgustinov, T. Ekman, N. Ongkingco, and J. Tibble, "Generative and transformational techniques in software engineering ii," R. Lämmel, J. Visser, and J. a. Saraiva, Eds. Berlin, Heidelberg: Springer-Verlag, 2008, ch. .QL: Object-Oriented Queries Made Easy, pp. 78–133.

[13] T. Zhang, X. Zheng, Y. Zhang, J. Zhao, and X. Li, "A declarative approach for java code instrumentation," *Software Quality Control*, vol. 23, no. 1, pp. 143–170, Mar. 2015.

[14] R. Cattell, "Scalable sql and nosql data stores," *ACM SIGMOD Record*, vol. 39, no. 4, pp. 12–27, December 2010.

[15] H. T. Plugge Eelco, Membrey Peter, *The definitive guide to MongoDB : the noSQL database for cloud and desktop computing*. Apress Berkely, CA, USA, 2010.

[16] D. Crockford, "Json: The fat-free alternative to xml," in *Proc. of XML*, Boston, USA, December 2006.

[17] C. D. H.-H. James R. Cordy and E. Promislow, "Txl: A rapid prototyping system for programming language dialects," *Computer Languages*, vol. 16, no. 1, pp. 97–101, 1991.

[18] TXL, http://www.txl.ca/index.html, 2015, [Online; accessed 20-March-2015].

[19] J. Hunter, "Jdom makes xml easy," in *Sun's 2002 Worldwide Java Developer Conference*, 2002.

[20] A. Colmerauer and P. Roussel, "History of programming languages—ii," T. J. Bergin, Jr. and R. G. Gibson, Jr., Eds. New York, NY, USA: ACM, 1996, ch. The Birth of Prolog, pp. 331–367.

[21] G. Kniesel and U. Bardey, "An analysis of the correctness and completeness of aspect weaving," in *Proceedings of the 13th Working Conference on Reverse Engineering*, ser. WCRE '06. Washington, DC, USA: IEEE Computer Society, 2006, pp. 324–333.

[22] R. C. Holt, "Structural manipulations of software architecture using tarski relational algebra," in *Proceedings of the Working Conference on Reverse Engineering (WCRE'98)*, ser. WCRE '98. Washington, DC, USA: IEEE Computer Society, 1998, pp. 210–.

[23] P. Rademaker, "Binary relational querying for structural source code analysis," the Netherlands, 2008.

[24] P. Klint, "How understanding and restructuring differ from compiling " a rewriting perspective," in *Proceedings of the 11th IEEE International Workshop on Program Comprehension*, ser. IWPC '03. Washington, DC, USA: IEEE Computer Society, 2003, pp. 2–.

[25] M. Verbaere, E. Hajiyev, and O. De Moor, "Improve software quality with semmlecode: An eclipse plugin for semantic code search," in *Companion to the 22Nd ACM SIGPLAN Conference on Object-oriented Programming Systems and Applications Companion*, ser. OOPSLA '07. New York, NY, USA: ACM, 2007, pp. 880–881.

[26] D. Beyer, A. Noack, and C. Lewerentz, "Efficient relational calculation for software analysis," *IEEE Trans. Softw. Eng.*, vol. 31, no. 2, pp. 137–149, Feb. 2005.

[27] C. Noguera, C. D. Roover, A. Kellens, and V. Jonckers, "Program querying with a soul: The barista tool suite," in *Proceedings of the 2011 27th IEEE International Conference on Software Maintenance*, ser. ICSM '11. Washington, DC, USA: IEEE Computer Society, 2011, pp. 582–585.

[28] E. Hajiyev, M. Verbaere, and O. de Moor, "Codequest: Scalable source code queries with datalog," in *Proceedings of the 20th European Conference on Object-Oriented Programming*, ser. ECOOP'06. Berlin, Heidelberg: Springer-Verlag, 2006, pp. 2–27.

[29] F. Yamaguchi, N. Golde, D. Arp, and K. Rieck, "Modeling and discovering vulnerabilities with code property graphs," in *Proceedings of the 2014 IEEE Symposium on Security and Privacy*, ser. SP '14. Washington, DC, USA: IEEE Computer Society, 2014, pp. 590–604.

[30] J. Webber, "A programmatic introduction to neo4j," in *Proceedings of the 3rd Annual Conference on Systems, Programming, and Applications: Software for Humanity*, ser. SPLASH '12. New York, NY, USA: ACM, 2012, pp. 217–218.

[31] R.-G. Urma and A. Mycroft, "Source-code queries with graph databaseswith application to programming language usage and evolution," *Science of Computer Programming*, vol. 97, pp. 127–134, 2015.