Mining Stack Overflow for Discovering Error Patterns in SQL Queries

Csaba Nagy and Anthony Cleve
PReCISE Research Center, University of Namur, Belgium
{csaba.nagy,anthony.cleve}@unamur.be

***Abstract*—Constructing complex queries in SQL sometimes necessitates the use of language constructs and the invocation of internal functions which inexperienced developers find hard to comprehend or which are unknown to them. In the worst case, bad usage of these constructs might lead to errors, to ineffective queries, or hamper developers in their tasks.**

This paper presents a mining technique for Stack Overflow to identify error-prone patterns in SQL queries. Identifying such patterns can help developers to avoid the use of error-prone constructs, or if they have to use such constructs, the Stack Overflow posts can help them to properly utilize the language. Hence, our purpose is to provide the initial steps towards a recommendation system that supports developers in constructing SQL queries.

Our current implementation supports the MySQL dialect, and Stack Overflow has over 300,000 questions tagged with the MySQL flag in its database. It provides a huge knowledge base where developers can ask questions about real problems. Our initial results indicate that our technique is indeed able to identify patterns among them.

***Index Terms*—SQL, Mining Stack Overflow, Error Patterns, Code Clones, Recommendation Systems**

I. Introduction

Stack Overflow (SO) is the main Q&A website of Stack Exchange Network, where software developers can discuss computer programming-related questions. It was publicly launched in 2008[1] and since then it has become one of the most popular Q&A sites. Today, as of June 2015, the site has 4.4 million users who asked 9.6 million questions with an answer rate of 74%. Its daily traffic is over 8 million visits[2]. The popularity of Stack Overflow means that it now has a large knowledge base of several programming topics and it also attracts researchers.

In this paper, we introduce an approach to mine error patterns in SQL queries extracted from SO posts. Our goal is to organize questions into groups based on the identified patterns, so a recommendation system can use these patterns to search whether a query contains language constructs that repeatedly appear on SO and directly point to questions with similar constructs.

Existing mining techniques for Stack Overflow usually analyze the textual content [1], [2]. There are some approaches for Java that can be utilized to exploit *island parsing* and use extra information gathered from the code blocks to support *recommendation systems* [3]. These techniques extend the analysis of textual content with extra information obtained from e.g., variable names, API usage and stack traces. The initial success of these techniques encouraged us to take a closer look at SQL code blocks in SO questions.

SQL is a *declarative programming language* and differs from procedural languages in many aspects. It treats *set*s as its fundamental data structure and it permits the construction of a complex query in one single statement by using different language constructs or by invoking native functions of the underlying database management system. The nature of the language requires a special way of thinking and this may be hard to learn for less experienced developers. Moreover, complex structures might lead to error-prone, ineffective statements that should be avoided. Today there are around 300,000 questions on the site tagged with the MySQL tag, while C++ has 375,000 and Java has 874,000.

Our technique focuses on the code blocks of the questions and tries to identify common patterns among the SQL statements within these blocks. We parse the extracted queries with a robust parser and run a pattern detection algorithm on the resulting *abstract syntax tree* (AST) of the parser. The output is a set of patterns which regularly appears in questions and their related groups of posts.

To evaluate our technique and its usefulness in practice, we sought answers to the following research questions:

- *RQ1: Is it possible with our approach to identify common language constructs (patterns) among SQL statements in Stack Overflow questions?*
- *RQ2: Do these constructs indicate the usage of SQL that might be error-prone, inefficient or hard to comprehend?*
- *RQ3: Can we improve the pattern detection by considering duplicates of SO questions?*

II. Related Work

In the past few years, several studies have been conducted on mining Stack Overflow. The popularity of the topic led to SO being the target of the mining challenge of MSR in 2013[3] and 2015[4].

The closest to our approach is described in the study by Ponzanelli et al., who proposed an *island parsing* technique to construct a *heterogeneous abstract syntax tree* (H-AST) for analyzing SO discussions about Java [3]. Their H-AST include Java constructs, stack traces, XML/HTML documents and JSON fragments. They also implemented a recommender

[1] http://www.joelonsoftware.com/items/2008/09/15.html
[2] http://stackexchange.com/sites?view=list\#traffic
[3] http://2013.msrconf.org/challenge.php
[4] http://2015.msrconf.org/challenge.php

system as an Eclipse plugin (*Prompter*), which generates Stack Overflow queries from code context in the IDE. Prompter exploits textual similarity, and code or API usage similarity. It is an advanced version of their first prototype tool *Seahawk* [4], which analyzed the textual content.

Beyer et al. also tried to identify common development issues, but they adapted a manual approach [5]. Barua et al. and Lineras-Vásquez employed automatic approaches to study trends and common problems that developers have [1], [2]. They relied on LDA to analyze the textual content of the posts.

The target of error pattern mining is not limited to SO. E.g., Thummalapenta et al. mined error patterns from source code [6], while Livshits et al. mined common error patterns in revision histories [7]. There are also several methods available for mining API usages [8], [9], [10] and using the information gathered by recommender systems [11], [12]. A comparison of source code mining techniques was presented by Khatoon et al. [13].

Our approach is different from the existing approaches as (1) we target SQL code fragments in Stack Overflow questions and (2) SQL, as a declarative language differs in nature from the languages targeted by existing approaches. The novelty of our mining technique also comes from the application of AST-based clone detection to code fragments in the questions.

III. Background

Stack Overflow may be regarded as a forum for computer programming questions. A user can ask a question from the community and mark one answer as accepted out of all the answers given by the experts. Users can vote as well, resulting in better quality answers with higher scores that are more likely to be accepted, while poor answers will get less attention and may be removed later. Questions with more upvotes represent better-explained problems and are more likely to be important for the community.

Questions can be tagged in order to keep them organized. For instance, a question about a MySQL query in PHP can be marked with the *php* and *mysql* tags so it appears for the users who are interested in these topics. A sample question of a typical optimization problem can be seen in Figure 1.

Although SO does not tolerate people repeatedly asking the same question and recommends that its users check if the same question has already been asked, users do not always realize that their problem might have been raised before. Moreover, it can be hard for an inexperienced developer to recognize that a problem is a specific instance of a more general one (which was probably asked before on the site). Members with a good reputation can mark a question as a duplicate of another, but the question may have some answers before it gets the attention of the moderators. Hence, the same problem can appear several times on the site.

The fact that users tend to ask similar questions inspired books in this area, e.g., as Bill Karwin says in his SO profile[5], "*I've written a book, SQL Antipatterns: Avoiding the Pitfalls of Database Programming from Pragmatic Bookshelf, based on the most common SQL problems I've answered on Stack Overflow and other forums, mailing lists, and newsgroups over the past 15 years*".

[5]http://stackoverflow.com/users/20860/bill-karwin

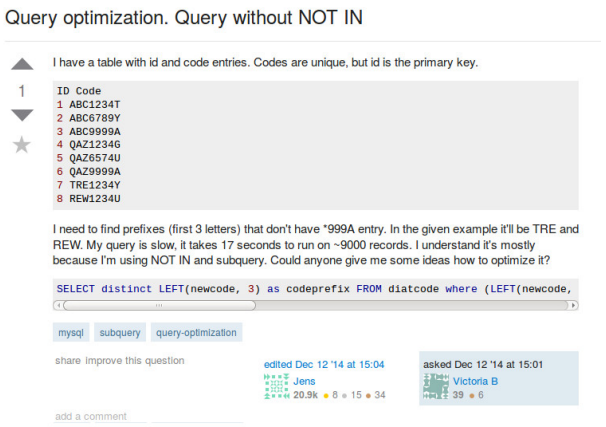

Figure 1. A sample Stack Overflow question with a *mysql* tag. There are over 200 questions on SO of the optimization problem of (NOT) IN subqueries.

For us, a question represents a coding issue and it potentially contains some problematic SQL code fragments, while answers hopefully contain solutions for this issue. Recommendation systems usually rely on this fact and help developers, e.g., by searching keywords from the coding context in the questions, and bringing the features of SO closer to the developer in their IDE [14], [4]. Our plan with a recommendation system is to carry on in this direction and to recognize if a pattern (which came up on SO) appears in a query (which might be embedded in the application code) and display the related posts.

By problematic code fragments we do not necessarily mean code with certain errors (e.g. syntactical). Questions on SO can describe different kinds of problems and according to the Android study of Beyer et al. [5] 'Error?' or 'What is the problem?' type questions represent 40% of the questions, while the rest are those like 'How to?', 'Is it possible?', 'Why?'. Code blocks in these questions are problematic in the sense that the user who asked about them encountered some issues or difficulties with them, e.g. a performance issue that can be seen in Figure 1.

IV. Pattern Detection in SQL Statements

Figure 2 gives an overview of the main steps of our approach, which we will now elaborate on.

A. Processing the Stack Overflow Dump

The data dump of Stack Overflow is published by Stack Exchange in XML format[6], and it contains all the relevant data needed for our analysis. The dump is divided into 'smaller' XML files. The one that describes the posts was about 29 Gb on Sept 14, 2014 and it contained over 25M posts (questions and answers). Below, we will rely on the data gathered from this dump.

[6]http://archive.org/details/stackexchange – user contributions of SO are licensed under *cc-by-sa 3.0*; see http://creativecommons.org/licenses/by-sa/3.0/

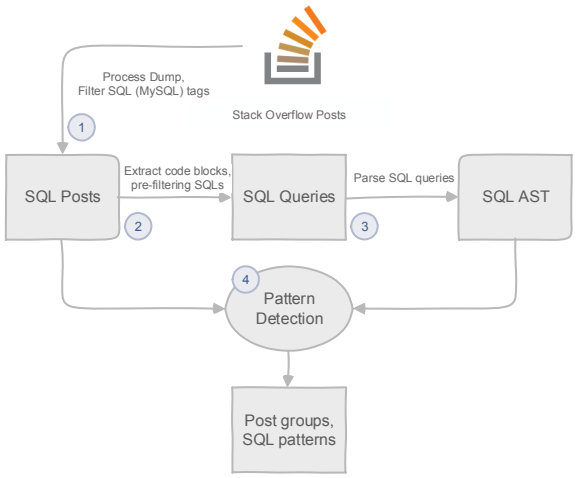

Figure 2. The main steps of the approach: first we process the dump of Stack Overflow posts and filter questions tagged with the *mysql* tag (1), then we extract code blocks containing SQL statements (2), which we then parse in the next step to produce an AST (3) for pattern detection (4).

Not all the posts are necessarily relevant to us, so first we filter them so as to just work with those posts that were marked with the *mysql* tag. (Our parser is currently only able to handle the MySQL dialect, but additional dialects could be considered here as well.) For this filtering, we load the XML, identify the questions with the *mysql* tag, and keep only these questions and their related posts (answers). After applying this filtering approach, we got a 1.1 Gb XML file, with 271,117 questions and 500,607 answers in it.

B. Extracting Code Blocks, Pre-filtering SQLs

Bodies of Stack Overflow questions are written in Markdown and code blocks are placed between the `<code>` and `</code>` tags. So for this step, we simply extract these code blocks. Notice, however, that users sometimes use the code blocks for formatting purposes (see the block of the sample data in Figure 1). In order to just work with the blocks that are relevant to us, we apply some additional filtering. We keep only those which contain SQL keywords (we check to see if *select*, *insert*, *update*, *delete*, *create*, *alter*, and so on appear in the block). Ideally, we should have only SQL blocks later, but some non-SQL fragments are likely to get through the filtering phase, which is acceptable here, as our parser drops them anyway if it cannot process them. For the example in Figure 1, it means that we just keep the last code block with the select statement and the filter will drop the first one which is only used to present some sample data.

Besides this pre-filtering of SQL code blocks, we apply character encoding transformations. The XML is in 'utf-8' encoding, and posts widely use the Unicode character set which can cause problems later on for the parser if unhandled characters appear outside of string literals, for instance.

C. Parsing SQL Queries

After extracting the SQL queries, the next step is to parse them and construct their ASTs. For this purpose, we use our robust SQL parser introduced in our previous paper [15], where we implemented a concept location approach to identify SQL queries in Java applications. The pattern detection problem is different from the concept location problem, where we had to match just one concrete query to others extracted from a Java source code base.

Our parsing approach can be viewed as an island-parsing technique, but with a slightly different goal. An island parser typically parses some recognizable structures (the *islands*) in a text and does not care about the rest (the *water*). In our case, we parse only the code blocks of the questions and we seek to build a complete AST for statements in these blocks. However, these blocks sometimes contain some non-SQL text, which makes some parts of the code unrecognizable for the parser. For instance, imagine typos in the text (e.g. 'form' instead of 'from') or a typical situation when someone types '...' instead of a complete field list of a select statement. These unrecognizable code parts represent the water in our case. Hence, we have huge islands with some water in between. As our main goal is to find patterns in the structures of the queries, we wish to keep the original query structure, so we insert special nodes as placeholders to the AST in the place of the unrecognized code parts.

```
SELECT t1.a, count(*) FROM t1, ... GROUP BY t1.a
```

Figure 3. A SQL query containing a code fragment ('...') that would normally cause a syntactic error.

Figure 3 shows a sample query extracted from SO which contains an unrecognized code fragment ('...') and Figure 4 shows its corresponding AST.

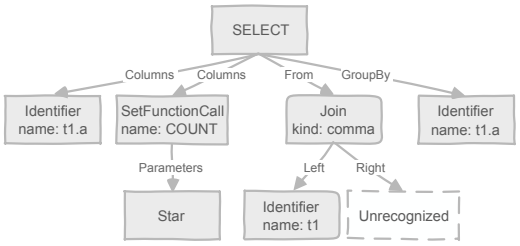

Figure 4. AST of the sample query in Figure 3.

D. Pattern Detection

To detect patterns in SQL statements, we implemented an AST-based clone detection algorithm based on the approach introduced by Baxter et al. [16]. Their clone detection algorithm first puts the AST nodes into buckets based on a hash function and identifies matching subtrees in each bucket with the help of a similarity function. As they say, "*This allows the straightforward detection of exact sub-tree clones. If we hash sub-trees to B buckets, then only those trees in the same bucket need be compared, cutting the number of comparisons by a factor of B.*" [16]

The main difference between our approach and theirs is that we need to handle the placeholders of unrecognized code fragments in the AST. Hence, our tree matching must be more permissive. The main steps of our clone detection algorithm are as follows:

1) *Preparation*: For each t_i AST node, we calculate the size of its subtree (as the number of nodes in it) and check see if it contains unrecognized fragments.
2) *Buckets*: We put all the nodes in buckets whose subtree is larger than $mintree$ or contains an unrecognized code fragment.
3) *Tree matching*: For each bucket B_k we identify matching subtrees.
4) *Keep only maximum clones*: For each clone pair (t_i, t_j), if the parents of t_i and t_j are clones, we remove (t_i, t_j) from the clone list (and keep the parents).
5) *Create clone classes*: if (t_i, t_j), and (t_i, t_z) or (t_j, t_z) are clones, we put t_i, t_j and t_z into the same clone class.

The *preparation* step is necessary to filter subtrees which do not reach a minimum number of nodes that we consider for pattern detection. We also need this minimum ($mintree$) for performance reasons and to keep larger patterns in focus instead of smaller ones (e.g., simple function calls). This step requires one preorder traversal of the AST.

To group nodes in *buckets*, we use the node kind of the AST node. This way, we have separate buckets for different kinds of nodes and we can eliminate unnecessary comparisons between any two subtrees with different types of root nodes. Notice that this could be improved by generating hashes that combine the node kinds and the attributes of the nodes.

The *tree-matching* algorithm is the core of the algorithm and it consists of the following steps:

1) For each (t_i, t_j) where t_i and t_j are trees in bucket B_k, invoke $match(t_i, t_j)$:
   a) return True, if t_i and t_j are the same nodes, or one of them is a node representing unrecognized fragments.
   b) return False, if the attributes of t_i and t_j are different, except for node kinds where we ignore attributes (identifiers or literals).
   c) for all subtree edges $e(t_i, t_{i_e})$ call $match(t_{i_e}, t_{j_e})$ where t_{i_e} is the child of t_i following the edge e and t_{j_e} is the child of t_j following the corresponding edge. Return True if (and only if) all the children match.

In the next step, we remove clone pairs whose parents are also clones, so we *keep only maximum clones*; and lastly, we organize the clone pairs into *clone classes*. One clone class will represent a pattern at the end of the process.

V. Evaluation

We evaluated our approach on the database export of SO from Sept 14, 2014. As an outcome of applying the first filtering steps, we extracted 271,117 questions, which were marked with the *mysql* tag. From these questions, we extracted 565,001 code blocks which contained keywords of SQL statements. Our parser was able to process 239,461 SQL statements out of these code blocks. Notice that the code blocks may (and usually do) contain non-SQL code such as Java or PHP code, so based on these results we could not make an estimate on the success rate of the parser. We manually investigated a random sample of 100 code blocks where the parser failed to produce the AST for the statements, and we found that 66 contained PHP code, 10 Java, 1 C++, 1 stored procedure, 5 syntactically incorrect SQLs, and in 17 cases there was sample data or other non-SQL text in the code block. The results show, however, that we managed to extract 0.88 statements on average for SO questions of MySQL.

Due to space limitations, here we present a brief summary of the results and we have made all the data (extracted posts, queries and patterns) available as an *online appendix*[7].

RQ1: Can our approach identify common language constructs (patterns) among SQL statements in Stack Overflow questions?

We executed our pattern detection algorithm with a minimum threshold of 10 nodes and identified a total number of 33,988 patterns after considering all the different node kinds in the AST. Table I lists some statistics (number of patterns originating from the node kind; average/max no. of pattern instances, tree size, and no. of questions) concerning the patterns of some selected node types (see the online appendix for more detailed statistics).

Table I
Statistics on the patterns of selected node kinds

Node Kind	No. of Patterns	Instances		Tree Size		SO Questions	
		Avg	Max	Avg	Max	Avg	Max
BinaryExpression	615	10.09	3681	17.80	247	8.10	3096
BinaryQuery	524	13.07	378	27.14	388	8.26	198
CaseExpression	375	5.05	70	15.37	60	2.67	31
Delete	198	3.67	30	19.31	55	3.46	30
FunctionCall	40	4.10	17	16.08	46	2.48	12
Insert	1206	5.87	274	21.62	339	4.15	194
Join	3866	16.42	10080	24.79	395	14.96	8953
NativeFunctionCall	1339	5.13	157	17.16	169	3.05	113
Select	7898	4.16	206	24.28	1182	3.99	205
SelectExpression	3824	7.00	618	21.89	353	5.86	545
SetFunctionCall	324	10.24	243	14.22	81	4.98	106
UnaryExpression	1707	7.08	289	17.65	165	5.29	243
Update	882	4.89	508	18.84	109	4.40	370

As an answer, the approach is able to identify patterns among the SQLs embedded in questions. The large number of patterns may be beneficial for a recommendation system, as it is more likely that it will be able to point the user to related problems. But it might also be a problem if it points to patterns which do not indicate error-prone usage of SQL. Hence, we need to take a closer look at these patterns by addressing RQ2.

RQ2: Do these constructs indicate usage of SQL that might be error-prone, inefficient or hard to comprehend?

To answer RQ2, we manually investigated the patterns. Namely, by sorting the list of patterns by the number of instances, we see that the most problematic node kinds are the joins. Just a simple inner join with two aliased tables appears in 8,953 SO questions, and without aliases it appears in 6,231 posts. Joining tables is definitely not a bad usage of SQL, but it is one of the first steps to constructing complex queries, which might be harder to understand for inexperienced developers.

The pattern with the biggest number of AST nodes has 1,182 nodes and appears in 2 questions[8][9] by the same author,

[7] http://perso.unamur.be/~cnagy/icsme2015-era/
[8] http://stackoverflow.com/questions/6847738/slow-query-with-in-operator
[9] http://stackoverflow.com/questions/6862743/need-to-extend-my-query

where the developer has a slow query with an IN operator listing lots of literals. The questions were not marked as duplicates, although they could have been, and our algorithm identified them.

```
SELECT @rownum := @rownum + 1 AS rank, student_id, gpa
    FROM students
```

Figure 5. Reoccurring pattern of using variables to solve ranking of selected records in MySQL

There are several patterns that could be spotted by experts as well. Figure 3 shows an example of using variables to have a ranking value for each row. Since MySQL has no ranking functionality this is a common way of overcoming this drawback, but use of variables like this requires caution as "*the order of evaluation for expressions involving user variables is undefined*"[10].

In general, after a manual investigation, it should not be concluded that the patterns are error-prone. Nevertheless, some of them may indicate incorrect usage of SQL and outliers clearly demonstrate this. It is probably advisable to further filter these results or to combine the technique with other approaches, such as the analysis of the textual content.

RQ3: Can we improve the pattern detection by considering duplicates of SO questions?

Considering only patterns where at least one question has a duplicate, results in a set of 1,360 patterns here. This filtering might be reasonable as we get a list of patterns which appear among questions that were already marked as duplicates on the site. It seems to confirm our view that these language constructs appear in problems regularly asked by developers.

VI. Research Opportunities

Our approach for identifying common patterns in SQL queries of Stack Overflow posts is an initial step towards a recommendation system and we evaluated it with this purpose in mind, but the results already open several research challenges and opportunities.

Among others, the current result set of patterns remains too large to expose general error-patterns, such as the SQL AntiPatterns of Karwin [17]. However, our parsing and matching algorithms constitute a sound basis in this direction. Extending our approach with ranking heuristics for instance might provide a list of error-prone patterns that could then be sorted into categories by SQL experts.

Other studies have tried to model how questions and answers are accepted on SO and how developers vote on questions. Our pattern detection technique could also be used to support such a model by identifying similarities among upvoted/downvoted posts or among accepted/unaccepted answers. What is more, it could pinpoint duplicated questions if desired.

Also, currently we just analyze the code fragments of the questions. Combining this with the analysis of the textual content (both in the questions and in the answers) would open up many other possibilities.

[10]https://dev.mysql.com/doc/refman/5.7/en/user-variables.html

VII. Conclusions

The huge knowledge base of Stack Overflow contains similar questions that address the same issues and not just among those questions that were marked as duplicates. Some of these similarities can be found manually. E.g., one can search on SO for the problem in Figure 1 as '`"NOT IN" optimize [mysql]`' and one will get over 200 posts, while the ineffective use of subqueries with `NOT IN` can be usually optimized with the proper use of a join or with the 'exists strategy'[11].

Our initial results indicate that our automatic technique for analyzing the SQL statements of the code blocks in questions really can identify patterns, and these patterns may be helpful for recommendation systems. We note, however, that to limit the analysis to just the code structure might produce noise, so a combined approach where the textual content is taken into account is advisable here.

References

[1] A. Barua, S. Thomas, and A. Hassan, "What are developers talking about? An analysis of topics and trends in Stack Overflow," *Empirical Software Engineering*, vol. 19, no. 3, pp. 619–654, 2014.

[2] M. Linares-Vasquez, B. Dit, and D. Poshyvanyk, "An exploratory analysis of mobile development issues using Stack Overflow," in *Proc. of MSR 2013*, May 2013, pp. 93–96.

[3] L. Ponzanelli, A. Mocci, and M. Lanza, "StORMeD: Stack Overflow Ready Made Data," in *Proc. of MSR 2015*. ACM, 2015.

[4] L. Ponzanelli, A. Bacchelli, and M. Lanza, "Seahawk: Stack Overflow in the IDE," in *Proc. of ICSE 2013*. IEEE, 2013, pp. 1295–1298.

[5] S. Beyer and M. Pinzger, "A manual categorization of Android app development issues on Stack Overflow," in *Proc. of ICSME 2014*, Sept 2014, pp. 531–535.

[6] S. Thummalapenta and T. Xie, "Alattin: Mining alternative patterns for defect detection," *Automated Software Engg.*, vol. 18, no. 3-4, pp. 293–323, Dec. 2011.

[7] B. Livshits and T. Zimmermann, "Dynamine: Finding common error patterns by mining software revision histories," *SIGSOFT Softw. Eng. Notes*, vol. 30, no. 5, pp. 296–305, Sep. 2005.

[8] Y. Lamba, M. Khattar, and A. Sureka, "Pravaaha: Mining Android applications for discovering API call usage patterns and trends," in *Proc. of the 8th India Software Engineering Conference (ISEC2015)*. ACM, 2015, pp. 10–19.

[9] T. Xie and J. Pei, "MAPO: Mining API usages from open source repositories," in *Proc. of MSR 2006*. ACM, 2006, pp. 54–57.

[10] H. Kagdi, M. L. Collard, and J. I. Maletic, "An approach to mining call-usage patternswith syntactic context," in *Proc. of the 22nd IEEE/ACM Int'l Conf. on Automated Software Engineering (ASE2007)*. ACM, 2007, pp. 457–460.

[11] M. Bruch, T. Schäfer, and M. Mezini, "On evaluating recommender systems for API usages," in *Proc. of RSSE 2008*. ACM, 2008, pp. 16–20.

[12] J. Cordeiro, B. Antunes, and P. Gomes, "Context-based recommendation to support problem solving in software development," in *Proc. of RSSE 2012*. IEEE, 2012, pp. 85–89.

[13] S. Khatoon, G. Li, and A. Mahmood, "Comparison and evaluation of source code mining tools and techniques: A qualitative approach," *Intell. Data Anal.*, vol. 17, no. 3, pp. 459–484, May 2013.

[14] A. Bacchelli, L. Ponzanelli, and M. Lanza, "Harnessing Stack Overflow for the IDE," in *Proc. of RSSE 2012*. IEEE, 2012, pp. 26–30.

[15] C. Nagy, L. Meurice, and A. Cleve, "Where was this SQL query executed? A static concept location approach," in *Proc. of SANER 2015*, March 2015, pp. 580–584.

[16] I. D. Baxter, A. Yahin, L. Moura, M. Sant'Anna, and L. Bier, "Clone detection using abstract syntax trees," in *Proc. of ICSM 1998*. IEEE Comp. Soc., 1998, pp. 368–377.

[17] B. Karwin, *SQL Antipatterns: Avoiding the Pitfalls of Database Programming*, 1st ed. Pragmatic Bookshelf, 2010.

[11]https://dev.mysql.com/doc/refman/5.6/en/subquery-optimization.html