Parsing and Analyzing SQL Queries in Stack Overflow Questions

Csaba Nagy, Anthony Cleve
`{csaba.nagy,anthony.cleve}@unamur.be`

PReCISE Research Center, University of Namur, Belgium

1 Introduction

Stack Overflow (SO) is the main Q&A website of Stack Exchange Network, run by Stack Exchange Inc, where software developers can discuss computer programming questions. It was publicly launched in 2008[1] and since then it has become one of the most popular Q&A sites. Within 2 years, the site reached 300,000 registered users and more than 7 million monthly visits (as of August 2010) [MMM$^+$11] and these numbers still grow rapidly. In September 2014, the site had 3.5M users [LP15] while today, after half a year, it has 4.2M users who have asked 9.3M questions with an answer rate of 74%. Its daily traffic is over 8 million visits a day[2].

This rapid growth and increasing popularity of Stack Overflow made it a large knowledge base of several programming topics which also attracts researchers. To mention a few examples, they study actual trends that developers follow [BTH14], design questions of Q&A systems [MMM$^+$11], island parsing techniques to analyze posts [LP15], recommendation systems [RYR14, PBL13, CAG12], and try to model the quality of the posts [PMB$^+$14, PMBL14].

In our paper, we introduce an approach to parse and analyze SQL queries in Stack Overflow questions with the main goal to identify common error patterns among them. Such similar structures in SQL statements can point to problematic language constructs (e.g., antipatterns) which should be avoided by developers.

Stack Overflow does not tolerate to ask the same question more than once, and recommends its users to double-check before sending a question whether it has been already asked or not. Users, however, cannot always realize that their problem was asked before. Moreover, it can be very hard for a non-experienced developer to recognize that a problem is just a specific instance of a more general one. Members with more reputation can mark a question as a duplicate of another, but even this way the question has been already asked and may have some answers as well before it gets to the attention of moderators. Hence, the same problem can show up several times on the site and patterns potentially exist among them. This fact already inspired books in this area, e.g., as Bill Karwin says it in his SO profile[3], "*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*" [Kar10].

2 Overview of the Approach

2.1 Introduction

SO can be considered as a forum for computer programming questions. A user can ask a question from the community and mark one answer as accepted of all the answers given by the experts. Users can vote as well, resulting that the answers with better quality have higher scores and 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, get more attention and are more likely to be important for the community. Users can also gain reputation scores and badges based on their activity and on the scores of their posts.

Questions are tagged in order to keep them organized. For instance, a question about querying a MySQL database in PHP can be marked with the *php* and with the *mysql* tags so it shows up for the users who are interested in these topics.

[1]`http://www.joelonsoftware.com/items/2008/09/15.html`
[2]`http://stackexchange.com/sites?view=list\#traffic`
[3]`http://stackoverflow.com/users/20860/bill-karwin`

For us, a question represents a coding issue and potentially contains some problematic SQL code fragments, while answers contain solutions for this issue. Recommendation systems rely on this fact and help developers, e.g., by searching for keywords in the questions, and bringing the features of the site closer to the developers in their IDE [BPL12, PBL13]. These techniques mostly work with NLP or LDA [LP15], and island-parsing techniques [BPL12, PBL13].

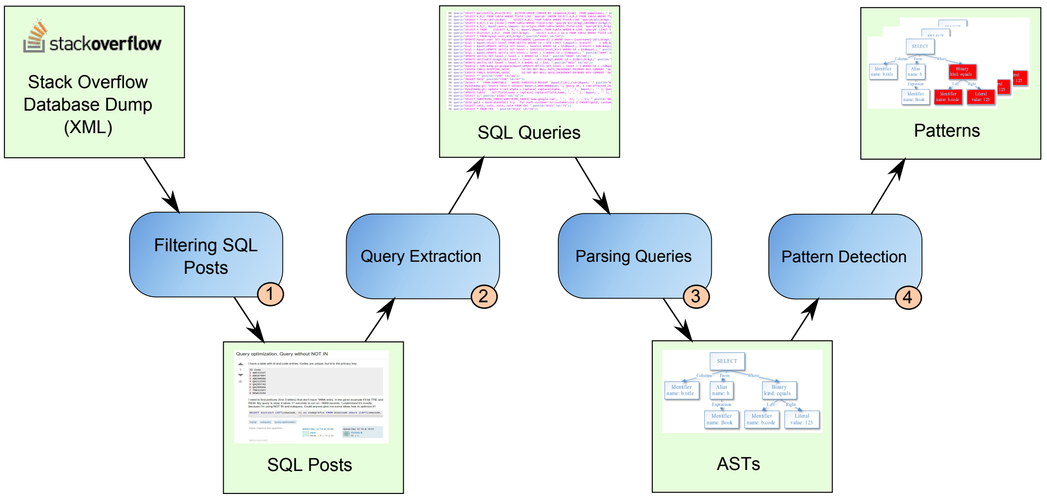


Figure 1: Main steps of the approach

Figure 1 represents an overview of the main steps of our approach, which can be considered as an automatic technique where we analyze the SQL code fragments in the questions and try to identify error patterns among them.

2.2 Filtering SQL Posts

The data dump of Stack Overflow is published by Stack Exchange in XML format[4], which contains all the relevant data for our analysis. All user contributions of SO are licensed under cc-by-sa 3.0[5].

The database dump is divided into smaller parts, and the size of the XML which contains the posts was about 29 Gb on September 14, 2014 (which we use in the rest of the paper). It contained over 25M posts (answers and questions both).

First, we filter this database to cover only posts that were marked with the *mysql* tag, because our parser is currently only able to deal with MySQL dialect. This step can be simply done by processing the XML and extracting the questions which belong to this tag, and extract their related answers too.

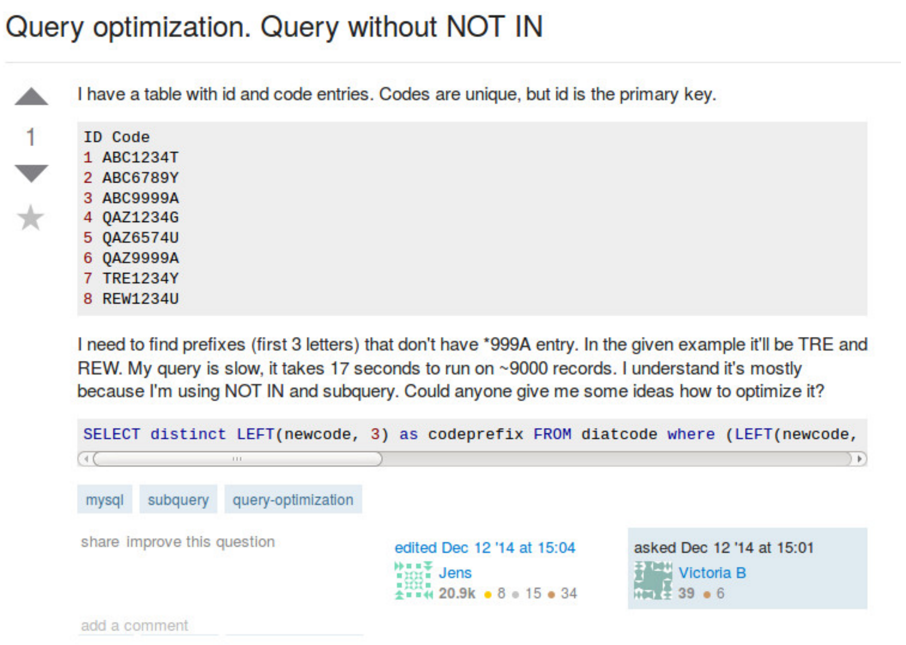
As a result of the filtering, we got a 1.1 Gb XML file, with about 271,117 questions and 500,607 answers.

2.3 Query Extraction

Once we have the posts which are related to SQL issues, it is possible to extract from these questions the code blocks which contain SQL statements. Here, we extract statements only from the questions (and not from the answers), as these statements are more likely to contain problematic language constructs.

A sample SO question with the *mysql* tag, can be seen in Figure 2. After a title and a short description, this example has a code block containing a select statement. Stack Overflow questions are written in Markdown and code blocks are placed between `<code>` and `</code>` tags. So for this step, we simply extract these code blocks and apply some additional filtering in order to keep blocks where we find keywords of SQL statements, which we can parse actually (we check if *select*, *insert*, *update*, *delete*, *create*, *alter*, etc. appears in the block). This filtering is important, because code blocks tagged as MySQL may still contain non-SQL (e.g., Java, PHP) code too. Ideally, we should have only SQL code blocks later, but some non-SQL codes are likely to get through the filtering, which is still acceptable here, as the parser will drop them anyway if it won't be able to process them. For the example in Figure 2, it means that we keep only the last code block with the select statement and the filter will drop the first one which is used only to present some sample data.

[4]`http://archive.org/details/stackexchange`
[5]`http://creativecommons.org/licenses/by-sa/3.0/`


Query optimization. Query without NOT IN

I have a table with id and code entries. Codes are unique, but id is the primary key.

```
ID Code
1 ABC1234T
2 ABC6789Y
3 ABC9999A
4 QAZ1234G
5 QAZ6574U
6 QAZ9999A
7 TRE1234Y
8 REW1234U
```

I need to find prefixes (first 3 letters) that don't have *999A entry. In the given example it'll be TRE and REW. My query is slow, it takes 17 seconds to run on ~9000 records. I understand it's mostly because I'm using NOT IN and subquery. Could anyone give me some ideas how to optimize it?

```
SELECT distinct LEFT(newcode, 3) as codeprefix FROM diatcode where (LEFT(newcode,
```

Figure 2: A sample Stack Overflow question with a *mysql* tag

As a result of this step, we had 564,941 code blocks containing SQL keywords, which is about 2 code blocks per question on average.

2.4 Parsing SQL Queries

After extracting the SQL queries from the questions, the next step is to parse the statements and construct their ASTs. For this purpose, we use our robust SQL parser introduced in our previous work [NMC15].

Our approach can be seen 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 want to have a complete AST for statements in these blocks. However, these blocks usually contain some non-SQL text too, which makes some parts of the code unrecognizable for the parser. E.g. imagine typos in the text (e.g., 'form' instead of 'from') or a typical situation when someone writes '...' 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 just some little water among them. Also, our goal is to find patterns in the structures of the queries, so we want to keep their original structure. For this reason, we insert special ('joker') nodes to the AST in the place of the unrecognized code parts.

As a result of this phase, currently, we were able to construct ASTs for 167,992 SQL statements, which means about 0.62 statements on average for the questions with the *mysql* tag.

2.5 Pattern Detection

The final step is to detect common patterns among the extracted statements. This step is under development at the moment but we have initial algorithms from our previous work [NMC15], where we implemented a tree-matching algorithm to locate the position in the Java source code, where a given query was constructed and sent to the database. The pattern searching problem is similar, but more general. In the matching problem of the concept location task, we had to match a concrete query to several queries extracted from a Java code base. However, it was not always possible to determine the full SQL statements in Java, that is, we had to match a concrete AST to several ASTs containing 'joker' nodes. Here, we have to find patterns as similar subtrees among ASTs containing 'joker' nodes.

3 Conclusions and Future Plans

Our research work is currently in an early stage and we could report here the main idea with some early results. We implemented the first steps (data preprocessing/filtering, query extraction and parsing) for MySQL, and we still need to improve and evaluate our pattern detection algorithm. However, the results from these first, parsing phases are already promising as we are able to extract over 167k SQL statements which are related to MySQL coding issues. We expect promising results from the pattern detection algorithms too. There are similar questions addressing the same issues in the database and not just among the questions which were marked as duplicates. Some of these similarities can be easily spotted in the database manually, for example, one can search for the problem in Figure 2 as '`"NOT IN" optimize [mysql]`' and it gives over 200 results, while an ineffective use of `NOT IN` with a subquery, can be simply avoided by the proper use of joins. Initial results show that automatic techniques are able to identify error patterns in this huge knowledge base which will be useful for developers working with SQL.

References

[BPL12] Alberto Bacchelli, Luca Ponzanelli, and Michele Lanza. Harnessing Stack Overflow for the IDE. In *Proceedings of the Third International Workshop on Recommendation Systems for Software Engineering (RSSE2012)*, pages 26–30. IEEE Press, 2012.

[BTH14] Anton Barua, Stephen W. Thomas, and Ahmed E. Hassan. What are developers talking about? An analysis of topics and trends in Stack Overflow. *Empirical Software Engineering*, 19(3):619–654, 2014.

[CAG12] Joel Cordeiro, Bruno Antunes, and Paulo Gomes. Context-based recommendation to support problem solving in software development. In *Proceedings of the Third International Workshop on Recommendation Systems for Software Engineering*, RSSE '12, pages 85–89. IEEE Press, 2012.

[Kar10] Bill Karwin. *SQL Antipatterns: Avoiding the Pitfalls of Database Programming.* Pragmatic Bookshelf, 1st edition, 2010.

[LP15] Michele Lanza Luca Ponzanelli, Andrea Mocci. StORMeD: Stack Overflow ready made data. In *Proceedings of the 12th Working Conference on Mining Software Repositories (MSR2015).* ACM Press, 2015.

[MMM+11] Lena Mamykina, Bella Manoim, Manas Mittal, George Hripcsak, and Björn Hartmann. Design lessons from the fastest Q&A site in the west. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems (CHI2011)*, pages 2857–2866. ACM, 2011.

[NMC15] Csaba Nagy, Loup Meurice, and Anthony Cleve. Where was this SQL query executed? a static concept location approach. In *Proceedings of the 22nd International Conference on Software Analysis, Evolution and Reengineering (SANER2015)*, pages 580–584, March 2015.

[PBL13] Luca Ponzanelli, Alberto Bacchelli, and Michele Lanza. Seahawk: Stack Overflow in the IDE. In *Proceedings of the 2013 International Conference on Software Engineering (ICSE2013)*, pages 1295–1298. IEEE Press, 2013.

[PMB+14] Luca Ponzanelli, Andrea Mocci, Alberto Bacchelli, Michele Lanza, and David Fullerton. Improving low quality Stack Overflow post detection. In *Proceedings of the 2014 IEEE International Conference on Software Maintenance and Evolution*, ICSME '14, pages 541–544. IEEE Computer Society, 2014.

[PMBL14] Luca Ponzanelli, Andrea Mocci, Alberto Bacchelli, and Michele Lanza. Understanding and classifying the quality of technical forum questions. In *Proceedings of the 14th International Conference on Quality Software (QSIC2014)*, pages 343–352, Oct 2014.

[RYR14] M.M. Rahman, S. Yeasmin, and C.K. Roy. Towards a context-aware IDE-based meta search engine for recommendation about programming errors and exceptions. In *Proceedings of the Software Evolution Week - IEEE Conference on Software Maintenance, Reengineering and Reverse Engineering (CSMR-WCRE 2014)*, pages 194–203, Feb 2014.