Using Observed Behavior to Reformulate Queries during Text Retrieval-based Bug Localization

Oscar Chaparro, Juan Manuel Florez, Andrian Marcus
The University of Texas at Dallas, Richardson, TX, USA
{ojchaparroa, jflorez, amarcus}@utdallas.edu

***Abstract*—Text Retrieval (TR)-based approaches for bug localization rely on formulating an initial query based on a bug report. Often, the query does not return the buggy software artifacts at or near the top of the list (*i.e.*, it is a *low-quality* query). In such cases, the query needs reformulation. Existing research on supporting developers in the reformulation of queries focuses mostly on leveraging relevance feedback from the user or expanding the original query with additional information (*e.g.*, adding synonyms). In many cases, the problem with such *low-quality* queries is the presence of irrelevant terms (*i.e.*, noise) and previous research has shown that removing such terms from the queries leads to substantial improvement in code retrieval. Unfortunately, the current state of research lacks methods to identify the irrelevant terms. Our research aims at addressing this problem and our conjecture is that reducing a *low-quality* query to only the terms describing the Observed Behavior (OB) can improve TR-based bug localization. To verify our conjecture, we conducted an empirical study using bug data from 21 open source systems to reformulate 451 *low-quality* queries. We compare the accuracy achieved by four TR-based bug localization approaches at three code granularities (*i.e.*, files, classes, and methods), when using the complete bug reports as queries versus a reduced version corresponding to the OB only. The results show that the reformulated queries improve TR-based bug localization for all approaches by 147.4% and 116.6% on average, in terms of MRR and MAP, respectively. We conclude that using the OB descriptions is a simple and effective technique to reformulate *low-quality* queries during TR-based bug localization.**

***Index Terms*—Observed Behavior, Query Reformulation, Bug Localization, Text Retrieval**

I. Introduction

Automated Text Retrieval (TR) has been widely used by researchers to develop techniques to support developers during bug localization [31, 40–42, 47, 49, 52, 56–63]. These techniques are instances of TR-based concept/feature location in source code [20, 38] and TR-based traceability link recovery [16], and tackle bug localization as a document retrieval problem. TR-based bug localization approaches rely on formulating an initial query based on a bug report (*i.e.*, its title/summary and description). Often, the query does not return the buggy code artifacts at the top of the list (*e.g.*, in the top-10 results). In such cases, the query is considered of *low-quality* and needs reformulation [25, 39]. Existing research on supporting developers in the reformulation of *low-quality* queries focuses mostly on leveraging relevance feedback from the user [21], pseudo-relevance feedback based on previous search results [25], or using additional information to replace or expand the query (*e.g.*, adding synonyms) [37, 45, 51]. However, in many cases, the problem with such queries is the presence of irrelevant terms (*i.e.*, noise) and previous studies [11] have shown that removing these terms from the queries (*i.e.*, *query reduction*) leads to substantial improvement in code retrieval. Unfortunately, the current state of research lacks methods to identify the irrelevant terms.

Our research aims at addressing this problem by leveraging the type of natural language information used in bug reports. Existing research [10, 14, 64] found that one of the main types of natural language content in bug reports is the Observed Behavior (OB). OB describes the current (mis)behavior of the software, which is generally deemed to be incorrect or unexpected. A recent study found that users frequently report the OB in bug descriptions (*i.e.*, in ~93% of the cases) [10], which underlines its important role in describing a software bug. Based on these findings and our experience in bug report analysis and TR-based bug localization, we conjecture that *the description of the OB has fewer noisy terms than the rest of the bug report, hence reducing a low-quality query to the terms describing the OB can improve TR-based bug localization*. The usage scenario we envision here is straightforward. Developers use the entire content of a bug report as a query (*i.e.*, both title/summary and description), and optionally other information leveraged by the TR-based technique they use – this is the typical scenario in TR-based bug localization [3, 20]. If the buggy code document is not identified in the top part of the returned results, then the developer selects the OB from the bug report and uses it as the new query.

This paper presents an empirical study aimed at verifying our conjecture. Based on existing data sets in TR-based bug localization and software testing research [28, 40, 59, 63], we sampled a set of bug reports and relevant change data from 78 versions of 21 open source systems. These bug reports result in *low-quality* queries, that is, they return the first buggy code document below the top-10 of the ranked list of source code documents [25]. We manually analyzed these *low-quality* bug reports and observed that 451 contain OB. We compare the accuracy achieved by four TR-based bug localization approaches (*i.e.*, BugLocator [63], BRTracer [59], Lobster [40], and Lucene [26]) at three code granularities (*i.e.*, files, classes, and methods) when using the complete bug reports as queries versus a reduced version corresponding to the OB only. The results show that the reduced queries improve TR-based bug localization for all four approaches and all granularities by 147.4% (MRR) and 116.6% (MAP),

on average. The results support our conjecture, which means that developers can reformulate a *low-quality* query by simply selecting the part that describes the OB and expect better retrieval results.

The main contributions of this paper are the following: (1) empirical evidence that using OB descriptions to reformulate *low-quality* queries improves the accuracy of four TR-based bug localization approaches; this simple and efficient query reformulation method requires no additional information to the bug description; and (2) a data set of labeled bug reports and queries that can be used for replication purposes and for future research in TR-based bug localization [9].

II. Background and Related Work

In this section, we briefly describe the main TR-based bug localization (TRBL) approaches and we discuss existing work on query reformulation in the context of source code retrieval.

A. TR-based Bug Localization

TRBL techniques are specific instances of TR-based concept/feature location in source code [20, 38] and TR-based traceability link recovery [16], which formulate bug localization as a document retrieval problem. A bug report is used as query to search a document space built from source code artifacts of a software system and retrieve a list of code documents (*e.g.*, files, classes, functions, or methods) relevant to the query. The relevance of a source code document to a query is determined by the textual similarity between them: the higher the textual similarity the more likely the document is to contain the bug. The targeted code documents are the ones that contain the bug described in the bug report. What differentiates TR-based bug localization from the more general code retrieval approaches is the use of bug reports as queries. TRBL techniques often use additional information, related to the current bug report, to adjust the ranking of the relevant code documents. Additional information leveraged by existing TRBL techniques includes: code version history [52, 57, 58, 62], similar bug reports [15, 49, 57–59, 62, 63], stack traces [40, 58, 59, 62], code structure [2, 49, 57, 58, 62], or combinations of the above [49, 57–59, 62].

We focus the discussion in this section on approaches designed specifically for bug retrieval (*i.e.*, they use information from or related to bug reports), rather than more generic concept/feature location and traceability link recovery approaches, which could also be used for bug localization. All TRBL approaches follow a common process, consisting of:

1) Building a corpus using the source code of the software.
2) Indexing the corpus using a TR model.
3) Formulating an initial query based on the bug report.
4) Ranking the documents with respect to the query, based on the TR model used and additional information related to the bug report.
5) Inspecting the retrieved documents. If the buggy code document is found, the process ends.
6) Reformulating the query if the buggy code is not identified, and resuming the process at step 4.

Previous research in concept/feature location and traceability link recovery focused on improving all the six steps of this process and TRBL techniques utilize much of that research. The main research efforts in TRBL focused primarily on step 4, especially by leveraging additional information related to the bug reports, as mentioned above.

Software history information is used by TRBL approaches to boost code artifacts with high defect/change probability based on code change records (*e.g.*, version control records). The code artifacts boosted are those found in change-sets that were intended to fix bugs. The boost amount can depend on different factors, *e.g.*, the number of times a code artifact has been fixed [52, 58, 62] or how long ago this happened [52, 57].

Bug fix history is also used to complement textual similarity. A set of previously fixed bug reports is kept, each one with its corresponding *fix-set*: the set of code documents that were modified in order to fix the bug. A query (*i.e.*, the current bug report) is compared to each previously-fixed bug report. The documents in each *fix-set* are boosted according to some criteria, *e.g.*, the textual similarity of the fixed bug with the query [15, 49, 57–59, 62, 63].

Bug reports sometimes contain stack traces, which are also used to alter the text-based ranking. Some TRBL approaches work on the assumption that the buggy code artifacts could be directly referenced by these traces, and use regular expressions to identify referenced classes/files [40, 58, 59, 62]. The set of suspicious classes/files is expanded by identifying artifacts (in)directly referenced in the code of the ones found in the stack trace. These relationships can be found by using the system's call graph [40] or the files' import statements [59, 62].

Some approaches exploit query and document structure by simply splitting the query into two parts (bug report title and description) and the document in four (classes, methods, variables, and comments). Besides the score calculated from the full text of both the query and the document, additional scores are calculated from the similarities between each of the two query components and each of the document components (8 additional scores in total), and then all scores are added together. This assigns a greater weight to terms appearing in multiple fields of a document, increasing their discriminating power for retrieval [2, 49, 57, 58, 62].

Unlike this rich body of existing work in improving TRBL, our focus is on helping developers reformulating a *low-quality* query when needed (*i.e.*, step 6 in the above process).

B. Query Reformulation in TRBL and Code Retrieval

Existing research highlighted the challenges that developers face when (re)formulating queries for code retrieval [4, 13, 54]. On one hand, TRBL approaches mitigate the problems associated with formulating an initial query by utilizing the bug report [11]. On the other hand, existing research provides little or no guidance on what parts of the bug reports to use when reformulating a *low-quality* query [29].

Four general query reformulation strategies are found in the literature, namely, *query expansion* [6], *query replacement* [23, 24], *term selection* [29, 46], and *query reduction* [34]. *Query*

expansion consists in adding alternative terms (or phrases) to a query; *query replacement* changes (part of) a query with a new set of terms; *term selection* selects a set of terms from a query as a new query; and *query reduction* focuses on removing query terms.

Most existing research on *query reformulation* in code retrieval (including TRBL) has focused on *query expansion*. The methods to determine the alternative terms include relevance feedback from developers [21]; pseudo-relevance feedback [25, 53], which leverages the lexicon of the previous top code documents retrieved; the use of English or software ontologies (*e.g.*, WordNet) [51], which contain related terms to the ones in a query (*e.g.*, synonyms); or co-occurring term information from various software sources, such as source code, Stack Overflow (SO) questions, or regulatory documents [17, 37, 45]. Similar techniques have been applied in the context of code search, where the initial queries are reformulated based on thesauri (*e.g.*, lexical databases from SO) [22, 32, 33], pseudo-relevance feedback from SO results [43], co-occurrence and frequency of query terms with previous results and source code [27, 48], and textual similarity between the query and Application Programming Interfaces (APIs) [35].

Query replacement has been utilized mostly for traceability link recovery [23], where the terms from similar web and domain-specific documents to the query are leveraged to select a set of candidate terms to replace the initial query. Another query replacement method is by learning frequent terms from existing requirement-regulation trace corpora, and use them as the new query [24].

Little research has focused on *term selection*. Kevic *et al.* [29] recommended the top three terms in a change request that have the highest predictive power to retrieve the relevant code documents (*i.e.*, in top-10 of the list). Their findings suggest that terms that appear in both the summary and description of change requests are good candidates to be used as query [29]. In another work, Rahman *et al.* [46] leveraged term co-occurrences and syntactic dependencies to select the most important terms in a change request as a query. Related to *term selection*, other research focused on weighing terms (from the query) that occur in method names and calls [5] or terms corresponding to source code file names [18].

Regarding *query reduction*, recent research has shown that removing noisy terms from the query (*i.e.*, from bug reports) leads to substantial retrieval improvement in TR-based bug localization [11]. The few works that include some kind of *query reduction* rely on heuristics to remove the noisy terms. Specifically, Rahman *et al.* [45] discarded the terms different from nouns or those occurring in more than 25% of the code documents, since they are likely to be non-discriminating. Haiduc *et al.* [25] followed a similar strategy.

Both *term selection* and *query reduction* strategies result in reduced queries. Our OB-based query reformulation strategy implies *term selection*, as we are selecting and retaining part of the initial query in the reformulated query. On the other hand, we do not argue that all the terms we select are most relevant, but rather we conjecture that the terms we do not select are less relevant, hence we also do some form of *query reduction* (*i.e.*, noisy terms are being removed). To the best of our knowledge, this is the first work that investigates how the type of textual content from bug descriptions used as queries can improve TR-based bug localization via query reformulation.

III. Empirical study design

We performed an empirical study to verify our conjecture. We followed an evaluation process commonly used in TR-based concept/bug/feature location research [3, 20], using four data sets from previous bug localization and software testing work. We detail the design of the study in this section. The results of the evaluation are discussed in Section IV.

A. Context and Research Question

The *goal* of our empirical evaluation is to determine whether or not *low-quality* queries reduced to the terms describing OB improve the accuracy of TRBL approaches. The *context* of our study is represented by 78 versions of 21 open-source projects written in Java, which vary in size and domain. The data used in our study was extracted from four existing data sets, namely, Just *et al.*'s Defects4J testing data set (*a.k.a.* D4J) [28], Mills *et al.*'s data set on query quality assessment (*a.k.a.* QQ) [39], Moreno *et al.*'s bug localization data set (*a.k.a.* LB) [40], and Wong *et al.*'s bug localization data set (*a.k.a.* BRT) [59].

D4J is a collection of real bugs from five open source systems[1], that includes test cases, the buggy source code, and bug fixes [28]. This data set was created for software testing research. We adapted it to the TRBL context. Specifically, we collected the buggy code methods that were changed to fix the bugs and the bug reports that describe each bug. D4J does not provide the bug reports directly. We manually inspected the commit messages (available in the D4J data) corresponding to the bug fixes to extract the IDs of the bug reports, which were used to collect the bug reports from each system's issue tracker. Unfortunately, for some bugs and systems, we were not able to trace their respective bug reports. In the end, we collected the bug reports of 124 bugs from three systems (*i.e.*, Lang, Math, and Joda-Time).

The remaining (original) data sets (*i.e.*, the LB, QQ, and BRT data sets) were used in existing TRBL research. The LB data set is composed of 974 bugs and their corresponding TRBL data (*i.e.*, the bug reports, buggy source code, fixed code artifacts) from 17 versions of 14 open source projects. We decided to discard the data corresponding to the muCommander system because the original issues are no longer available. We used 815 bug reports with TRBL data from this data set in total. The QQ data is composed of 278 bugs and their corresponding TRBL data from 15 versions of 12 open source projects[2]. This data set includes all the needed TRBL data, except the original bug reports used to create the queries.

[1]We used Defects4J v1.0.1, which originally contained five systems. This data is available at https://github.com/rjust/defects4j/releases

[2]The authors of the QQ data set shared with us an updated version of it. Therefore, the number of versions and systems are slightly different to the ones reported in the authors' publication [39].

Based on the provided queries and preprocessing heuristics we mined the bug trackers of the corresponding systems and were able to trace the bug reports of 241 QQ bugs, which correspond to 13 versions of 11 software projects. The BRT data set is composed of 3,459 bugs and their corresponding TRBL data from 3 open source systems. We decided to discard the data from the AspectJ system, as we observed inconsistencies about the ground truth files (*i.e.*, the fixed code artifacts) for 68.5% of the queries. For example, the files changed to fix the bug from AspectJ 141956[3] include the following ones: ".../PR141956/base/A.java", ".../PR141956/base/C.java", and ".../PR141956/inc1/C.java". These are new files added to the system and used to test the bug fix (according to the commit metadata[4]). The files are not meant to be in the buggy version of the system, yet they were found in the system's source code provided in BRT. In the end, we used 3,173 bugs from the Eclipse and SWT projects. The four data sets used in our study have different granularities, namely, method-level for D4J and QQ, class-level for LB, and file-level for BRT.

Overall, we collected 4,353 bugs and their corresponding bug reports, the changed files for each bug report (used to fix the respective bug), and the source code of the software systems, for 78 versions of 21 open-source projects. We created a software document corpus from the code of each software version (*i.e.*, one corpus per version), according to the granularity of each data set (*i.e.*, methods, classes, or files). The corpus was created by extracting the identifiers, comments, and string literals. The documents in the corpus and the queries were normalized by using standard preprocessing: identifier splitting (based on the camel case and underscore formats), special characters removal, common English stop words and Java keywords removal, and stemming [44].

For each bug report, we created an *initial query*, by concatenating the bug report's title and description. Other studies in TRBL have also used the full text of the bug report to formulate the query [40, 59, 63]. In this study, we define *low-quality* queries as queries that return the first buggy code document below the top-10 of the ranked list of retrieved documents. The rank threshold is derived from previous empirical evaluations of TRBL, which assess the performance of TRBL techniques on the top-10 results of the list [59, 63]. Similar thresholds have been used in the context of query quality assessment [25]. In order to identify the *low-quality* queries in our data set, we relied on the document rankings generated by Lucene (see Section III-D) for all 4,353 initial queries. For each query, we measured the rank of the first buggy code document retrieved. We retained the queries with such a rank greater than 10 and excluded the ones that did not lead to any relevant documents retrieved since they do not benefit from term removal. This process resulted in 1,592 *low-quality* queries (*i.e.*, 1,190 BRT, 139 QQ, 36 D4J, and 227 LB queries). The entire data set used in our study can be found in our replication package [9].

[3]https://tinyurl.com/kte4lqt

[4]The commit (5f6a6b12c5) is available at: https://tinyurl.com/kvqrh3j

In the context of our study, we formulate the following research question:

RQ*: Do low-quality queries reduced to the terms describing OB improve the accuracy of TR-based bug localization?*

B. Observed Behavior Identification

In order to answer our research question, the terms corresponding to the OB were identified for the bug reports that resulted in *low-quality* queries. The number of bug reports was manageable for manual analysis in all data sets except for BRT, because of the high number of Eclipse bug reports (*i.e.*, 1,174). Therefore, we randomly sampled 142 bug reports from the Eclipse project, and selected all the bug reports from the other systems and data sets. A total of 560 bug reports that resulted in *low-quality* queries were selected from the four data sets for OB identification. Two people (*a.k.a.* coders – both authors of this paper) conducted sentence-level qualitative *coding* [50] on all 560 bug reports. The starting coding framework was defined by one of the coders by studying 25 issues from Davies *et al.*'s work [14]. The goal of this task was to analyze the issue descriptions in order to identify the sentences that corresponded to the OB. This task resulted in the initial coding criteria. Once the pilot study was completed, this person trained the other coder in a 45-minute session that involved discussing the coding results and some ambiguous sentences.

We summarize some of the most important coding criteria (the full list can be found in our replication package [9]). The coding focused only on natural language content written by the reporters, as opposed to code snippets, stack traces, or logs. However, the natural language referencing this information may indicate OB. Such cases were allowed for coding. An example of this case is: "*When I click the button, I get the following error: ...*". Uninformative sentences such as "*The menu does not work*" are insufficient to be considered OB. There must be a clear description of the observed (mis)behavior of the software, *e.g.*, "*The menu doesn't open when I click the button*". Also, explanations of attached code to the bug reports or about the system's internal code are not considered OB.

To facilitate the coding process, we built an automated tool that splits the text into sentences and paragraphs. The tool uses the Stanford CoreNLP toolkit [36] and heuristics (*e.g.*, punctuation). To minimize subjectivity, each bug report was coded by both coders. For each bug report, the coders analyzed the report text and marked the sentences corresponding to OB. The coding process also discarded the issues that were not actual bug reports, but feature requests or enhancements (*i.e.*, 98 reports). In the end, 462 bug reports from our four data sets were inspected and coded (see Table I). Overall, the OB identification step required significant manual effort. However, in an actual usage scenario, a developer only needs to select the OB terms from a single bug report, which takes seconds.

We analyzed the reliability of the coding process performed by the two coders regarding the presence and absence of OB in the 462 bug reports. We measured the inter-coder agreement at sentence level using three well-known metrics,

Table I: Statistics of each data set.

Data set	Code granularity	# of systems[a]	# of coded bug reports	# of bug reports containing OB	# of *low-quality* reduced queries
D4J	Method	3 (52)	36	35 (97.2%)	35
QQ	Method	11 (13)	103	100 (97.1%)	100
LB	Class	13 (16)	199	197 (99.0%)	196
BRT	File	2 (2)	130	125 (96.2%)	125
Overall*		**21 (78)**	**462**	**452 (97.8%)**	**451**

* Overall numbers without repetitions, [a] in parenthesis, # of system versions.

namely, the Observed agreement, Cohen's kappa (k) [12], and Krippendorff's alpha (α) [30]. Overall, each of two coders inspected 10,902 sentences from the bug reports, and decided whether or not they described Observed Behavior (see Table II). Our analysis reveals high inter-coder agreement levels. The coders agreed on 10,526 sentences (*i.e.*, 96.6% observed agreement). The kappa and alpha measures range from 73.9% and 84.3% across the four data sets, which account for 80.3% overall agreement (*i.e.*, *substantial agreement* [55]). The coders obtained *almost perfect agreement* on the BRT data set and *substantial agreement* for the remaining ones [55].

Table II: Inter-coder agreement measurements.

Data set	Total # of sentences	# of sentences agreed on	Observed agreement	Cohen's kappa (k)	Krippendorff's alpha (α)
D4J	426	394	92.5%	73.9%*	73.9%*
QQ	1,670	1,588	95.1%	78.0%*	78.0%*
LB	5,340	5,157	96.6%	79.5%*	79.5%*
BRT	3,466	3,387	97.7%	84.3%**	84.3%**
Overall	**10,902**	**10,526**	**96.6%**	**80.3%***	**80.3%***

* substantial, and ** almost perfect agreement [55].

Overall, 229 bug reports (*i.e.*, 49.6%) had some type of disagreement. To solve the disagreements, we selected 16 bug reports that presented the highest number of disagreements. Both coders examined each conflicting sentence in these reports and determined, through discussion and mutual agreement, the correct label for the sentences (*i.e.*, either OB or non-OB). This activity generated a common framework to solve the disagreements in the next step. The remaining 213 bug reports were evenly distributed between the two coders, and each one solved the disagreements of their corresponding reports individually. If a coder had a doubt about the correct label of a conflicting sentence, both coders would discuss and solve each case. In the process, the coders recorded the causes for each disagreement. We found that the main causes were omissions of actual OB sentences (found in 50.7% of the reports), sentences that were too high-level or vague to be considered as OB (found in 20.0% of the reports), and sentences describing code snippets or the system's internal code rather than OB (found in 7.8% of the reports).

C. Query Reduction Strategy

After the coding was completed and conflicts were resolved, reformulated queries were generated from the coded bug reports. Our analysis of the 462 coded bug reports reveals that 97.8% (*i.e.*, 452) of reports contain explicit OB sentences (see Table I). This proportion is in line with the ones measured in other bug reports data sets [7, 14]. From the 452 bug reports containing OB, we created 451 *reduced queries*[5], by using only the terms from the sentences that were coded as OB.

[5]After preprocessing, the query from the PIG-1935 report was voided.

D. TR-based Bug Localization Approaches

We selected four TRBL approaches to run the initial and reduced queries, namely Lucene [26], Lobster [40], BugLocator [63], and BRTracer [59]. The goal is to assess if the OB-based query reduction technique is influenced by the TBRL approach.

Lucene [26] is a retrieval technique implemented in the open source library of the same name [1], which combines the standard information retrieval Boolean model and the Vector Space Model (VSM) to compute the similarity between a query and a document. Lucene is a technique that relies only on textual information to retrieve the relevant (buggy) documents at any granularity. Therefore, we used this approach on all four data sets.

Lobster [40] is a TRBL technique that leverages stack traces found in bug reports. It boosts the classes that appear in these traces and also their related classes by using the system's call graph. Lobster works at class-level granularity, therefore, we used it only on the LB data set. It is important to note that this approach only makes a difference on bug reports that actually contain stack traces. Hence, we measured its accuracy using a subset of *low-quality* queries in the LB data set. We used the original implementation of Lobster, which was provided by its authors [40].

BugLocator [63] is a TRBL approach that combines information from bug fix history and file length to boost certain corpus documents. This approach uses a record of previously-fixed bug reports to boost the files that were fixed as a result, according to the textual similarity of these reports to the query. Additionally, it boosts all corpus source files based on their length (*i.e.*, number of terms). BugLocator works a file-level granularity, therefore, we used it only on the BRT data set. We used our own implementation of BugLocator in this study, based on the description provided in its corresponding publication [63].

BRTracer [59] is an extension of BugLocator, which uses stack trace information from bug reports and source file segmentation to boost source code files retrieved by BugLocator. Similar to Lobster, this technique boosts the source code files that appear in the traces, and other files (or classes) that are used in their internal code. In addition, the files are segmented into smaller documents, and the highest textual similarity between the segments and the query is used as the similarity of the whole file. We used our own implementation of BRTracer, which was used on the BRT dataset.

We evaluated our implementations of BugLocator and BRTracer[6] on the data sets provided by their respective authors. On average, for BRTracer, the absolute differences of our results from those reported by the authors are 2.6%/1.0% for MRR/MAP (see Section III-E). For Buglocator, such differences are 4.4%/3.7% for MRR/MAP. We attribute these

[6]We implemented these approaches because their original code was not available at the time of conducting this study.

differences to variations in the exact preprocessing that we used versus the one applied by the authors (since the code corpus in the original data set does not come preprocessed). Hence, we do not expect large variations in the results when using the authors' implementation. We did not adapt these approaches to work at different corpus granularities, as it would have required major modifications to the original methods, resulting essentially in new TRBL approaches. Such changes are beyond the scope of our research since we are concerned with the impact of using the OB as queries on TRBL.

E. Accuracy Metrics

We compare the performance of the reduced queries against the initial queries using standard measures previously used in TR-based bug and feature location research [3, 20]. We perform the comparison only in the cases where a query could be successfully reduced (*i.e.*, the 451 queries) since otherwise, our approach would have no effect.

Effectiveness refers to the best rank obtained by any of the documents relevant to a query. We also compute the difference of effectiveness between the reduced and initial queries (*a.k.a.* rank difference).

Mean Reciprocal Rank (MRR) is a statistic that measures the quality of the ranking of a retrieval approach in a search task by capturing how close to the top of the result list a relevant (*i.e.*, buggy) document to a query q is retrieved. MRR is given by the average of the reciprocal *effectiveness* of a set of queries Q:

$$\text{MRR}(Q) = \frac{1}{|Q|}\sum_{q \in Q}\frac{1}{\text{effectiveness}(q)}$$

The higher the MRR value, the higher the ranking quality of the bug localization approach will be. MRR is an aggregate measure of how high the first relevant document ranks.

Mean Average Precision (MAP) is a measure of the accuracy of a retrieval approach based on the average precision of each query q in the set Q. Given R_q, the set of documents relevant to query q, the average precision is computed as the average of the precision values at the resulting rank of each document. MAP is the mean of the average precision of the set of queries Q, defined as follows:

$$\text{MAP}(Q) = \frac{1}{|Q|}\sum_{q \in Q}\frac{1}{|R_q|}\sum_{r \in R_q}\text{precision}(\text{rank}(r))$$

MAP reflects how well *all* the changed (*i.e.*, buggy) documents rank, in aggregate.

In order to understand in more details improvements or deteriorations in rankings, we need a finer-grained analysis and we also compute the followings metrics:

Number of queries improved and deteriorated. In our case, a query is *improved* if the effectiveness (*i.e.*, the rank of the first relevant document) of the reduced query is lower than the effectiveness of the initial query. A query is *deteriorated* if the effectiveness of the reduced query is higher than the effectiveness of the initial query. Otherwise, a query has *no effect*, *i.e.*, the effectiveness before and after applying the query reduction is the same. Ideally, we want a higher number of improved queries than deteriorated queries.

Improvement and deterioration magnitude. It measures the relative magnitude of improvement or deterioration, which is computed as $(r_b - r_a)/r_b$, *i.e.*, the change percentage of the rank of the first buggy artifact (*i.e.*, effectiveness) when using the reduced queries. We report the mean and median magnitude of improvement and deterioration. Ideally, we want the magnitude of improvement to be larger than the magnitude of deterioration. However, when comparing these magnitudes, one must proceed with caution. For example, if an initial query has effectiveness 20 and the reduced query improves it to 10, then the improvement magnitude is 50%. On the other hand, if the baseline effectiveness is 10 and the reduced query deteriorates it to 20, then the magnitude of the deterioration is 100%. Note that in both cases the absolute difference in effectiveness is the same (*i.e.*, 10).

Quality of a query. Comparing the magnitude of improvement and deterioration reveals only one aspect of the differences between two reformulation approaches. For example, assume that *approach A* achieves 75% improvement and *approach B* achieves 50% improvement, on average. It is easy to conclude that *A* is better than *B*. Now, let us assume that *A* improves queries that, on average, ranked at 400, so now they rank, on average, at 100, whereas *B* improves queries ranked at 40, on average, and now they are ranked at 20. One can conclude that *A* does not help much in practice (*i.e.*, the relevant documents are still ranked too far from the top of the list), whereas *B* has a significant positive impact. MAP partially addresses this issue, but it still provides no indication in which part of the ranking the improvements or deteriorations occur.

In order to analyze this aspect, we define four categories of query quality, depending on the queries' effectiveness when utilized by a TR-based bug localization approach. A query is considered *high-quality* (*i.e.*, Q_{10}) if its effectiveness is between 1 and 10, and *low-quality* if its effectiveness is greater than 10 (*i.e.*, Q_{10+}). We define three groups of *low-quality* queries. A query belongs to group Q_{20} if its effectiveness is between 11 and 20 (*i.e.*, *low quality*). If the effectiveness is between 21 and 30, then the query is labeled Q_{30} (*i.e.*, *lower quality*). Otherwise, when the effectiveness of the query is higher than 30, the query is labeled Q_{30+} (*i.e.*, *lowest quality*). Similar categories were defined in prior work on query reformulation [25]. Ideally, our goal is to convert *low-quality* queries into *high-quality* ones. At the same time, we want to avoid further deteriorating queries belonging to the Q_{20} group, while we are less concerned if Q_{30} and Q_{30+} queries get deteriorated. We report and analyze the number of queries that improve/deteriorate across categories (*e.g.*, from Q_{30+} to Q_{10}, or from Q_{20} to Q_{30}).

IV. Results and Discussion

We present and discuss the results produced by the initial and reduced *low-quality* queries using the four TRBL approaches on their corresponding data sets.

Table III: Mean Reciprocal Rank (MRR) and Mean Average Precision (MAP) of the initial and reduced queries.

Data set	TR-based approach	# of queries[a]	MRR			MAP		
			Initial queries	Reduced queries	Improv.	Initial queries	Reduced queries	Improv.
D4J	Lucene	30 (5)	2.7%	5.2%	94.9%	2.8%	5.3%	87.5%
QQ	Lucene	93 (7)	2.2%	6.4%	187.9%	2.0%	4.5%	127.5%
LB	Lucene	188 (8)	3.1%	12.9%	312.4%	2.8%	10.1%	258.6%
	Lobster	35 (1)	2.6%	10.7%	304.5%	2.2%	7.4%	235.5%
BRT	Lucene	123 (2)	2.8%	4.5%	61.3%	2.4%	4.1%	68.4%
	BugLocator	96 (2)	2.4%	3.2%	34.9%	2.0%	2.5%	20.3%
	BRTracer	89 (2)	2.2%	3.0%	36.0%	1.9%	2.2%	18.5%
Average			**2.6%**	**6.6%**	**147.4%**	**2.3%**	**5.2%**	**116.6%**

[a]. In parenthesis, number of (reduced) queries that did not retrieve the corresponding buggy code documents.

A. Overall TR-based Bug Localization Accuracy

Table III summarizes the results (in terms of MRR and MAP) obtained for each data set and TRBL approach, by utilizing the initial and reduced queries. The number of queries used varies across different TRBL approaches in the LB and BRT data sets. The reason for these differences is that different approaches lead to different sets of *low-quality* queries. This means that, after using the hybrid approaches (*i.e.*, Lobster, BugLocator, and BRTracer), some queries remain as *low-quality* queries, while others become *high-quality* and, therefore, are not included in the analysis for each approach.

Table III reveals that the reduced queries improve the baseline MRR and MAP values by different magnitudes depending on the approach and data set. The reduced queries improve the MRR values by 34.9% to 312.4%, and the MAP values by 18.5% to 258.6%, across all approaches and data sets. Analyzing the results across data sets, we observe that the highest improvement is for the LB data set, whereas the lowest improvement magnitude is achieved for the BRT data set. Remember that D4J and QQ are method-level granularity, LB is class-level, and BRT is file-level. Granularity-wise, for Java systems, we can consider class- and file-level to be somewhat similar. The different performance for Lucene across the LB/BRT data sets indicates that likely the corpus granularity does not seriously impact our reformulation technique. Approach-wise, BugLocator and BRTracer see the smallest improvement, while Lobster and Lucene see the most improvements. This indicates that BRTracer and BugLocator are less sensitive to noisy queries than Lucene and Lobster, yet OB-based query reduction still leads to significant improvements. Summarizing across different approaches and data sets, we found that, on average, the reduced queries improve the accuracy of TRBL by 147.4% and 116.6%, in terms of MRR and MAP, respectively. One can notice that the absolute MRR and MAP values for all the queries are low, which is to be expected given that all these are *low-quality* queries.

B. Accuracy across Query Categories

In order to understand where the MRR/MAP overall improvement comes from (*i.e.*, see Table III), we computed the TRBL accuracy across the categories defined in Section III-E for the initial queries and their reduced version. Table IV shows the relative magnitude of MRR and MAP improvements for the Q_{20}, Q_{30}, and Q_{30+} initial queries, respectively.

On average, we found that the OB-based query reduction strategy obtains the largest improvement magnitude for the *lowest quality* queries (*i.e.*, Q_{30+} queries; 229.4%/193% avg. MRR/MAP improvement). The *lower quality* queries (*i.e.*, Q_{30} queries) see somewhat less improvement (*i.e.*, 150%/129.9% avg. MRR/MAP). Finally, the better among the *low-quality* queries (*i.e.*, Q_{20} queries) see the lowest magnitude improvement (*i.e.*, 100.7%/64.5% avg. MRR/MAP). We conclude that the OB-based query reduction technique appears to work best with the *lowest quality* queries, yet it achieves significant improvements across all categories of *low-quality* queries.

Table IV: Mean Reciprocal Rank (MRR) and Mean Average Precision (MAP) of the initial Q_{20}, Q_{30}, and Q_{30+} queries and their corresponding reduced queries.

Query category	Average MRR			Average MAP		
	Initial queries	Reduced queries	Improv.	Initial queries	Reduced queries	Improv.
Q_{20} **queries**	7.0%	14.2%	100.7%	6.1%	9.9%	64.5%
Q_{30} **queries**	3.9%	10.0%	150.0%	3.5%	8.3%	129.9%
Q_{30+} **queries**	1.0%	3.6%	229.4%	1.0%	3.0%	193.0%

C. Trade-offs between Improved and Deteriorated Queries

As with any query reformulation technique, the improvement of some queries comes at the cost in the quality of others. The goal of any reformulation technique is to improve more queries than it deteriorates and the magnitude of improvement to be higher than the deterioration. While the MRR/MAP values indicate that this is the case with our approach, we perform a finer grained analysis in terms of the number of queries improved/deteriorated, and the average/median magnitude of improvement/deterioration, to understand the trade-offs.

Table V reveals that our reduction method improves some *low-quality* queries while deteriorating others across all data sets. The number of improved queries is higher than the number of deteriorated queries for the QQ and LB data sets. These numbers explain the high MRR and MAP values on these data sets, especially when using Lobster, where 25 queries (*i.e.*, 71.4%) are improved and only 8 are deteriorated (*i.e.*, 22.9%). The opposite situation is observed for the D4J and BRT data sets. The number of deteriorated queries is slightly higher than the number of improved queries, except when BRTracer is used, which leads to the same number of queries improved and deteriorated. We need to analyze the magnitude of the improvement/deterioration since the MRR/MAP improvements in these data sets indicate overall improvement. We observe that the average/median magnitude of deterioration is higher than the magnitude of improvement for all approaches and

Table V: Number of queries improved, deteriorated and unchanged, average (median) magnitude of improvement and deterioration, and average (median) rank difference when reducing the initial queries.

Data set	TRBL approach	Improved queries			Deteriorated queries			Unchanged queries
		#[a]	Avg. magn.[b]	Avg. rank diff.[b]	#[a]	Avg. magn.[b]	Avg. rank diff.[b]	#[a]
D4J	Lucene	11 (36.7%)	51.2% (64.6%)	44.5 (38.0)	16 (53.3%)	-222.1% (-155.4%)	-145.5 (-46.0)	3 (10.0%)
QQ	Lucene	50 (53.8%)	52.6% (53.9%)	675.0 (66.5)	34 (36.6%)	-353.4% (-66.1%)	-528.4 (-65.0)	9 (9.7%)
LB	Lucene	100 (53.2%)	61.7% (64.8%)	94.0 (29.0)	71 (37.8%)	-305.8% (-127.3%)	-133.1 (-47.0)	17 (9.0%)
	Lobster	25 (71.4%)	49.9% (45.8%)	86.1 (17.0)	8 (22.9%)	-46.3% (-38.4%)	-93.3 (-48.5)	2 (5.7%)
BRT	Lucene	54 (43.9%)	50.6% (50.4%)	425.2 (30.0)	61 (49.6%)	-680.3% (-143.2%)	-497.5 (-88.0)	8 (6.5%)
	Bug Locator	45 (46.9%)	40.6% (38.5%)	241.4 (37.0)	46 (47.9%)	-452.7% (-112.1%)	-407.2 (-85.5)	5 (5.2%)
	BRTracer	42 (47.2%)	39.9% (37.8%)	266.2 (39.0)	42 (47.2%)	-317.5% (-80.6%)	-386.5 (-85.5)	5 (5.6%)
Average		**46.7 (50.4%)**	**49.5% (50.8%)**	**261.8 (36.6)**	**39.7 (42.2%)**	**-339.7% (-103.3%)**	**-313.1 (-66.5)**	**7.0 (7.4%)**

In parenthesis: [a] percentage values, [b] median values.

data sets, except for Lobster on LB. The same situation is observed for the average/median rank difference between the initial and reduced queries. Summarizing across approaches and data sets, we found that the reduced queries achieve 49.5% (50.8%) average (median) improvement, *i.e.*, 261.8 (36.6) avg. (median) positions improved, with a significant deterioration cost: 339.7% in average (103.3% median), *i.e.*, 313.1 (66.5) avg. (median) positions deteriorated. To better understand these numbers, consider a query whose buggy code artifact ranks at position 20. Then, 49.5% (or 50.8%) improvement means that the artifact is re-ranked to position 10. On the other hand, 339.7% (or 103.3%) deterioration means that the artifact is re-ranked on position 88 (or 41). We require additional analysis to explain the overall MRR/MAP improvement, which we perform in the following subsection.

D. Queries Improved and Deteriorated between Categories

As we mentioned before, it matters what kind of queries get improved or deteriorated. For example, in an extreme case, we would prefer a reformulation approach that improves a single *low-quality* query from position 50 to 5, at the cost of deteriorating 100 queries from position 400 to 500 (*i.e.*, arguably, they are equally bad). Conversely, we would avoid an approach that improves 100 *low-quality* queries from position 500 to 400, yet it deteriorates one from 5 to 50. The metrics discussed above do not capture such cases. Generally, improving or deteriorating Q_{30+} or Q_{30} queries while maintaining them in the same Q_{30+} and Q_{30} categories matters less from a practical point of view (*i.e.*, really *low-quality* queries remain equally *low-quality*), whereas improving queries across categories matters most (*e.g.*, from Q_{30} to Q_{10} – *low-quality* queries become *high-quality*).

Table VI shows how many of the *low-quality* queries improved/deteriorated stayed within the same category and how many changed category. To be consistent with the acronyms in the table, we define a notation for the transitions between categories. When a Q_{20} query is improved and becomes *high-quality* (*i.e.*, Q_{10}), we denote it as $Q_{20\to10}$. When a Q_{20} query is deteriorated and becomes Q_{30}, we denote it as $Q_{20\to30}$. If a Q_{20} query is improved or deteriorated but it does not change its category, we denote is as $Q_{20\to20}$.

In our analysis, we focus on the following aspects:

- The best cases are when *low-quality* queries become *high-quality* (*i.e.*, $Q_{20\to10}$, $Q_{30\to10}$, or $Q_{30^+\to10}$). $Q_{30^+\to10}$ is the best-case scenario.
- The cases when a query improves to Q_{20} (*i.e.*, $Q_{30\to20}$, $Q_{30^+\to20}$, or $Q_{20\to20}$) are also desirable, although potentially less practical (*i.e.*, they are still *low-quality* queries).
- The remaining cases when there is improvement (*i.e.*, $Q_{30^+\to30}$, $Q_{30\to30}$ and $Q_{30^+\to30^+}$) are the least practical forms of improvement.
- Regarding deterioration, the worst-case scenario is when a Q_{20} query becomes Q_{30+} (*i.e.*, $Q_{20\to30}$+). These cases should be minimized.
- The cases when a Q_{20} query becomes Q_{30} or deteriorate within the same category (*i.e.*, $Q_{20\to30}$ or $Q_{20\to20}$) should be also minimized, although, they are not as severe as the previous ones.
- The remaining cases when there is deterioration (*i.e.*, $Q_{30\to30^+}$, $Q_{30\to30}$, or $Q_{30^+\to30^+}$) are the least problematic among the deterioration cases (*i.e.*, really *low-quality* queries become somewhat worse).

Table VI shows that our reduction strategy is able to improve *low-quality* queries into *high-quality* (*i.e.*, $Q_{*\to10}$) for all approaches and data sets. In the case of the LB data set, the number of queries that become *high-quality* is substantial. When Lucene and Lobster are used, 25% (*i.e.*, 47) and 28.6% (*i.e.*, 10) of the queries become *high-quality*, respectively. For the remaining approaches and data sets, such numbers are lower, *i.e.*, between 6.7% and 13.3% of the queries. It is worth noting that when Lucene is used on LB data, our reduction strategy is able to improve 24 (*i.e.*, 12.8%) Q_{30+} queries into *high-quality* (*i.e.*, $Q_{30^+\to10}$ – the best-case scenario). Fewer $Q_{30^+\to10}$ queries were obtained for the remaining approaches and data sets (*i.e.*, 1 or 2 queries). Summarizing across techniques and data sets, we found that, on average per approach and data set, our reduction strategy improves 14.8% *low-quality* queries into *high-quality*. This means that 14.8% of the *low-quality* queries are reduced and now ranked in the top-10 of the results list.

As mentioned before, the queries that improve into Q_{20} (*i.e.*, $Q_{*\to20}$) are also desirable. Table VI shows that the proportions of $Q_{*\to20}$ queries fall between 3.3% and 11.5% across approaches and data sets, with Lucene on D4J and QQ being the approach that achieves the lowest proportions (*i.e.*, 3.3% and 4.3%, respectively). On average, 8.9% *low-quality* queries were ranked higher, between positions 11 and 20, by our reduction strategy. We aggregate the number of queries that become *high-quality* and the ones that improve into Q_{20} (*i.e.*, $Q_{*\to\{10,20\}}$). These queries are the ones that most contribute to

Table VI: Number of queries of each transition between *high-* ($\mathbf{Q_{10}}$) and *low-quality* ($\mathbf{Q_{10+}}$) queries.

Data set	TRBL approach	Improved queries											
		Cross-category queries							Within-category queries			$\mathbf{Q_{*\rightarrow 20}}^a$	$\mathbf{Q_{*\rightarrow \{10,20\}}}^a$
		$\mathbf{Q_{20\rightarrow 10}}$	$\mathbf{Q_{30\rightarrow 10}}$	$\mathbf{Q_{30^+\rightarrow 10}}$	$\mathbf{Q_{30\rightarrow 20}}$	$\mathbf{Q_{30^+\rightarrow 20}}$	$\mathbf{Q_{30^+\rightarrow 30}}$	$\mathbf{Q_{*\rightarrow 10}}^a$	$\mathbf{Q_{20\rightarrow 20}}$	$\mathbf{Q_{30\rightarrow 30}}$	$\mathbf{Q_{30^+\rightarrow 30^+}}$		
D4J	Lucene	1	1	2	0	1	1	4 (13.3%)	0	0	5	1 (3.3%)	5 (16.7%)
QQ	Lucene	8	2	1	1	3	3	11 (11.8%)	0	2	30	4 (4.3%)	15 (16.1%)
LB	Lucene	11	12	24	6	11	6	47 (25.0%)	3	6	21	20 (10.6%)	67 (35.6%)
	Lobster	5	3	2	1	3	0	10 (28.6%)	0	1	10	4 (11.4%)	14 (40.0%)
BRT	Lucene	8	2	2	1	6	2	12 (9.8%)	5	3	25	12 (9.8%)	24 (19.5%)
	BugLoc.	6	0	2	3	3	0	8 (8.3%)	5	1	25	11 (11.5%)	19 (19.8%)
	BRTracer	4	0	2	3	2	0	6 (6.7%)	5	1	25	10 (11.2%)	16 (18.0%)
Average		**6.1**	**2.9**	**5.0**	**2.1**	**4.1**	**1.7**	**14 (14.8%)**	**2.6**	**2.0**	**20.1**	**8.9 (8.9%)**	**22.9 (23.7%)**

In parenthesis, $^a\cdot$ percentage values with respect to the total number of queries that retrieved a buggy code artifact.

Data set	TRBL approach	Deteriorated queries							
		Cross-category queries			Within-category queries			$\mathbf{Q_{20\rightarrow \{30,20\}}}^a$	$\mathbf{Q_{20\rightarrow *}}^a$
		$\mathbf{Q_{20\rightarrow 30}}$	$\mathbf{Q_{20\rightarrow 30^+}}^a$	$\mathbf{Q_{30\rightarrow 30^+}}$	$\mathbf{Q_{20\rightarrow 20}}$	$\mathbf{Q_{30\rightarrow 30}}$	$\mathbf{Q_{30^+\rightarrow 30^+}}$		
D4J	Lucene	0	4 (13.3%)	1	0	0	11	0 (0.0%)	4 (13.3%)
QQ	Lucene	2	1 (1.1%)	4	3	2	22	5 (5.4%)	6 (6.5%)
LB	Lucene	4	13 (6.9%)	10	4	0	40	8 (4.3%)	21 (11.2%)
	Lobster	1	0 (0.0%)	0	0	0	7	1 (2.9%)	1 (2.9%)
BRT	Lucene	4	9 (7.3%)	8	3	0	37	7 (5.7%)	16 (13.0%)
	BugLocator	1	3 (3.1%)	8	1	0	33	2 (2.1%)	5 (5.2%)
	BRTracer	1	4 (4.5%)	6	0	0	31	1 (1.1%)	5 (5.6%)
Average		**1.9**	**4.9 (5.2%)**	**5.3**	**1.6**	**0.3**	**25.9**	**3.4 (3.1%)**	**8.3 (8.2%)**

In parenthesis, $^a\cdot$ percentage values with respect to the total number of queries that retrieved a buggy code artifact.

the overall MRR/MAP performance from Table III. We can see higher proportions of $Q_{*\rightarrow\{10,20\}}$ queries for LB than for the other data sets (*i.e.*, 35.6% and 40% when Lucene and Lobster are used, respectively), which explain the high MRR/MAP values in such cases.

The Q_{20} queries that deteriorate to Q_{30} or deteriorate in the same category (*i.e.*, $Q_{20\rightarrow\{30,20\}}$) are undesirable. Table VI shows low proportions of such queries across approaches and data sets, *i.e.*, 3.1% on average. We aggregate the number of $Q_{20\rightarrow 30^+}$ and $Q_{20\rightarrow\{30,20\}}$ queries (*i.e.*, $Q_{20\rightarrow *}$). These queries are the ones that most impact the overall MRR/MAP performance from Table III. We can see lower proportions of these queries compared to $Q_{*\rightarrow\{10,20\}}$ queries (*i.e.*, the most desirable cases), across approaches and data sets, which explains the overall MRR/MAP improvements. On average, 8.2% of the Q_{20} queries are deteriorated. We now analyze the worst deterioration cases, that is, when Q_{20} queries become Q_{30+} (*i.e.*, $Q_{20\rightarrow 30^+}$). Table VI shows that the highest proportion of $Q_{20\rightarrow 30^+}$ queries is achieved by Lucene on D4J, LB, and BRT (*i.e.*, 13.3%, 6.9%, and 7.3%, respectively). The proportions of the remaining approaches and data sets fall between 1.1% and 4.5%, except for Lobster on LB, with zero $Q_{20\rightarrow 30^+}$ queries. Summarizing across techniques and data sets, we found that, on average, 5.2% Q_{20} queries are deteriorated into Q_{30+}, *i.e.*, 11% of deteriorated queries. The results indicate that, on average, most of the deteriorated queries (*i.e.*, 89%) are the least offensive type.

We manually inspected the initial and reduced queries, and the expected buggy code documents for the 34 $Q_{20\rightarrow 30^+}$ cases, which correspond to 30 unique queries (the overlap comes from the BRT data). We found three reasons for the deteriorations. The first (and main) reason is that important shared terms between the queries and the buggy code documents were removed from the initial queries (*i.e.*, found in 3 D4J, 1 QQ, 8 LB, and 8 BRT queries – 20 queries total). This happened when the terms appeared in sentences not describing OB. Then, the removal of the terms led to deterioration. For example, in the case of the bug report LANG-432[7], the term "***handling***" appears in the title, which does not describe OB. This term appears in the Javadoc comment of the buggy method `StringUtils:containsIgnoreCase(String, String)`: "*... Checks if String contains ...* ***handling*** *<code>null</code> ...*". By removing the term, the ranking of the buggy method deteriorated. The second reason was observed in queries from the Eclipse system only (*i.e.*, four BRT queries), which were generated from bug reports that contained stack traces. We observed that each one of such bug reports contained only one short OB sentence and a long stack trace (along with other non-OB content), which contained the buggy files (or the names of the implemented classes) in some of its entries. For instance, for Eclipse 94633[8], the reduced query comes from the bug report's title: "*Concurrent Modification Exception whilst editing Ant file*". Only the terms "***ant***" and "***file***" appear in the vocabulary of the buggy file **`AntElementNode.java`**, which occurs in the third entry of the stack trace: "*at ... AntElementNode ...* ***AntElementNode.java****:450)*". In this case, the removal of the stack trace and the few shared query terms lead to the deterioration. Finally, the third reason found is that non-OB sentences contained the name of the buggy artifacts (*i.e.*, found in 1 D4J and 5 LB queries – 6 queries total), as in the case of PIG-3310[9], in which several of its non-OB sentences contain the term `LOSplitOutput` (*i.e.*, the buggy class). Overall, we observed that the removed, yet important terms may appear in natural language sentences, stack traces, or code snippets. One way to improve our reformulation strategy

[7]https://issues.apache.org/jira/browse/LANG-432
[8]https://bugs.eclipse.org/bugs/show_bug.cgi?id=94633
[9]https://issues.apache.org/jira/browse/PIG-3310

is by identifying and including code snippets in the query along with the OB terms. Another situation was observed in the manually inspected bug reports. Some important terms in OB sentences also occurred in non-OB sentences. However, when the latter ones were removed to reduce the queries, the frequency of such terms in the new query decreased. This may contribute to the deterioration of our reformulation strategy. As future work, we plan to boost such terms in the reduced query, by increasing their frequency.

In conclusion, while Lucene obtains the highest MRR/MAP improvements across data sets, it also seems to be the most sensitive to query reduction, with higher absolute cost in terms of deterioration. The significant improvements (in some cases) offset the deterioration. As noted before, BRTracer and BugLocator achieve less MRR/MAP improvements than the other techniques but exhibit fewest offensive deteriorations.

V. Threats to Validity

We briefly discuss the threats that could affect the validity of our study. The main threat to *construct validity* is the subjectivity introduced in the construction of the labeled data set of bug reports (see Section III-B). To minimize subjectivity, we ensured that each bug report was coded by two coders independently. We also defined common coding criteria and trained the coders on them via interactive tutorials, which included examples and discussion of ambiguous cases. We also assessed coding reliability by measuring the inter-coder agreement using well-known metrics (see Section III-B).

In order to mitigate threats to the *conclusion validity* of our results, we compared the performance of the reduced and initial queries using standard measures (*i.e.*, effectiveness, MRR, and MAP), widely used in concept/feature location research [19] and in the evaluation of other TRBL approaches [40, 57, 59, 63]. As a further analysis, we evaluated the number of queries improved and deteriorated, and the relative magnitude of improvement and deterioration to perform a fine-grained analysis of the gain achieved by reducing the initial queries. Finally, we also grouped the queries in four categories (*high-quality* and *low-quality*, with *low-quality* further divided into three sub-categories) and analyzed the transition of queries among categories after reduction. Similar categories were defined in prior work on query reformulation [25].

The choice of context and TRBL approaches impacts the *internal validity* of our conclusions. The context of the study is represented by three data sets previously used in TRBL studies [39, 40, 59] and one used in software testing [28]. Such data sets have different granularity levels (*i.e.*, method-, class-, and file-level) and correspond to distinct Java software systems. While we observed variation in results across data sets and TRBL approaches, the common denominator in all treatments was the query reduction mechanism, which we consider the main factor in the observed improvements. Another threat to *internal validity* concerns the bug report sampling. To identify *low-quality* queries, we relied on the rankings generated by Lucene on those queries, and selected a random sample of (the corresponding) bug reports that was large enough to allow manual OB identification. Some queries, generated from this subset, turned out not being *low-quality* when retrieved using the other TRBL approaches (*i.e.*, Lobster, BugLocator, and BRTracer). This means that when evaluating our query reduction technique with these approaches, we used only queries that were *low-quality* for both Lucene and the respective approach, which indicates that some (other) *low-quality* queries for each approach may have been missed.

In order to strengthen *external validity*, we used 451 queries from bug reports from 78 versions of 21 different software systems, which are from different domains and sizes. While our set of bug reports is not small, a larger set of reports would have strengthened the results. Finally, we used four TRBL techniques, namely, Lucene [26], Lobster [40], BugLocator [63], and BRTracer [59]. The results may or may not vary when using different TRBL approaches.

VI. Conclusions and Future Work

We hypothesized that reducing bug reports to their Observed Behavior (OB) content can improve TR-based bug localization (TRBL) when the bug report forms a *low-quality* query. An empirical study on 451 such *low-quality* queries provided evidence in support of our conjecture. We observed that our query reformulation technique is robust to the corpus granularity and works best for the *lowest quality* queries, yet it achieves significant improvements across other types of *low-quality* queries. As with any query reformulation strategy, there is a trade-off between queries that improve and those that do not. We found that the trade-off is in favor of the improved queries, as most of the ones that deteriorate are of the lowest quality to begin with. Indirectly, we found that two of the approaches we experimented with (*i.e.*, BRTracer and BugLocator) are less sensitive to noisy queries than the other two (*i.e.*, Lucene and Lobster), while all benefit from the OB-based query reduction strategy. In conclusion, we consider the OB-based query reformulation strategy effective in improving TRBL, especially considering that it requires no additional information (than what is already in a bug report) and it demands minimal effort from the developer (*i.e.*, simply select the sentences describing the OB).

As for future work, we will investigate ways to further improve our query reformulation technique. Specifically, we will investigate boosting OB query terms that occur frequently in other parts of the bug report and including terms found in code snippets. We also plan to closely investigate other natural language descriptions contained in bug reports (*e.g.*, the expected behavior and the steps to reproduce the bug), and ways to automatically detect such contents to reformulate *low-quality* queries [8]. Finally, expanding the evaluation on more data sets, using more TRBL approaches, and conducting studies with end users is also planned.

Acknowledgments

This research was supported in part by the grant CCF-1526118 from the US National Science Foundation.

REFERENCES

[1] "https://lucene.apache.org/," 2017.

[2] N. Ali, A. Sabane, Y.-G. Gueheneuc, and G. Antoniol, "Improving bug location using binary class relationships," in *Proceedings of the International Working Conference on Source Code Analysis and Manipulation (SCAM'12)*, 2012, pp. 174–183.

[3] V. Arnaoudova, S. Haiduc, A. Marcus, and G. Antoniol, "The use of text retrieval and natural language processing in software engineering," in *Companion Proceedings of the International Conference on Software Engineering (ICSE'16)*, 2016, pp. 898–899.

[4] S. K. Bajracharya and C. V. Lopes, "Analyzing and mining a code search engine usage log," *Empirical Software Engineering*, vol. 17, no. 4-5, pp. 424–466, 2012.

[5] B. Bassett and N. A. Kraft, "Structural information based term weighting in text retrieval for feature location," in *Proceedings of the International Conference on Program Comprehension (ICPC'13)*, 2013, pp. 133–141.

[6] C. Carpineto and G. Romano, "A survey of automatic query expansion in information retrieval," *Computing Surveys*, vol. 44, no. 1, p. 1, 2012.

[7] O. Chaparro, "Improving bug reporting, duplicate detection, and localization," in *Proceedings of the International Conference on Software Engineering (ICSE'17)*, 2017, pp. 421–424.

[8] O. Chaparro, J. M. Florez, and A. Marcus, "On the vocabulary agreement in software issue descriptions," in *Proceedings of the International Conference on Software Maintenance and Evolution (ICSME'16)*, 2016, pp. 448–452.

[9] ——, "Replication package," 2017. [Online]. Available: https://seers.utdallas.edu/projects/ob-query-reformulation

[10] O. Chaparro, J. Lu, F. Zampetti, L. Moreno, M. Di Penta, A. Marcus, G. Bavota, and V. Ng, "Detecting missing information in bug descriptions," in *Proceedings of the Joint Meeting on Foundations of Software Engineering (ESEC/FSE'17)*, 2017, (to appear).

[11] O. Chaparro and A. Marcus, "On the Reduction of Verbose Queries in Text Retrieval Based Software Maintenance," in *Companion Proceedings of the International Conference on Software Engineering (ICSE'16)*, 2016, pp. 716–718.

[12] J. Cohen, "A coefficient of agreement for nominal scales," *Educational and psychological measurement*, vol. 20, no. 1, pp. 37–46, 1960.

[13] K. Damevski, D. Shepherd, and L. Pollock, "A field study of how developers locate features in source code," *Empirical Software Engineering*, vol. 21, no. 2, pp. 724–747, 2016.

[14] S. Davies and M. Roper, "What's in a Bug Report?" in *Proceedings of the International Symposium on Empirical Software Engineering and Measurement (ESEM'14)*, 2014, pp. 26:1–26:10.

[15] S. Davies, M. Roper, and M. Wood, "Using bug report similarity to enhance bug localisation," in *Proceedings of the Working Conference on Reverse Engineering (WCRE'12)*, 2012, pp. 125–134.

[16] A. De Lucia, A. Marcus, R. Oliveto, and D. Poshyvanyk, "Information retrieval methods for automated traceability recovery," in *Software and Systems Traceability*, J. Cleland-Huang, O. Gotel, and A. Zisman, Eds. Springer, 2012, pp. 71–98.

[17] T. Dietrich, J. Cleland-Huang, and Y. Shin, "Learning effective query transformations for enhanced requirements trace retrieval," in *Proceedings of the International Conference on Automated Software Engineering (ASE'13)*, 2013, pp. 586–591.

[18] T. Dilshener, M. Wermelinger, and Y. Yu, "Locating bugs without looking back," in *Proceedings of the International Conference on Mining Software Repositories (MSR'16)*, 2016, pp. 286–290.

[19] B. Dit, "Monitoring the searching and browsing behavior of developers in eclipse during concept location," M.Sc. Thesis, Wayne State University, 2009.

[20] B. Dit, M. Revelle, M. Gethers, and D. Poshyvanyk, "Feature location in source code: A taxonomy and survey," *Journal of Software: Evolution and Process*, vol. 25, no. 1, pp. 53–95, 2012.

[21] G. Gay, S. Haiduc, A. Marcus, and T. Menzies, "On the use of relevance feedback in ir-based concept location," in *Proceedings of the International Conference on Software Maintenance (ICSM'09)*, 2009, pp. 351–360.

[22] X. Ge, D. C. Shepherd, K. Damevski, and E. Murphy-Hill, "Design and evaluation of a multi-recommendation system for local code search," *Journal of Visual Languages & Computing*, 2016.

[23] M. Gibiec, A. Czauderna, and J. Cleland-Huang, "Towards mining replacement queries for hard-to-retrieve traces," in *Proceedings of the International Conference on Automated Software Engineering (ASE'10)*, 2010, pp. 245–254.

[24] J. Guo, M. Gibiec, and J. Cleland-Huang, "Tackling the term-mismatch problem in automated trace retrieval," *Empirical Software Engineering*, pp. 1–40, 2016.

[25] S. Haiduc, G. Bavota, A. Marcus, R. Oliveto, A. De Lucia, and T. Menzies, "Automatic query reformulations for text retrieval in software engineering," in *Proceedings of the International Conference on Software Engineering (ICSE'13)*, 2013, pp. 842–851.

[26] E. Hatcher and O. Gospodnetic, *Lucene in Action*. Manning Publications, 2004.

[27] E. Hill, M. Roldan-Vega, J. A. Fails, and G. Mallet, "Nl-based query refinement and contextualized code search results: A user study," in *Proceedings of the Conference on Software Maintenance, Reengineering, and Reverse Engineering (CSMR-WCRE'14)*, 2014, pp. 34–43.

[28] R. Just, D. Jalali, and M. D. Ernst, "Defects4j: A database of existing faults to enable controlled testing studies for java programs," in *Proceedings of the International Symposium on Software Testing and Analysis (ISSTA'14)*. ACM, 2014, pp. 437–440.

[29] K. Kevic and T. Fritz, "Automatic search term identification for change tasks," in *Companion Proceedings of the International Conference on Software Engineering (ICSE'14)*, 2014, pp. 468–471.

[30] K. Krippendorff, *Content Analysis: An Introduction to Its Methodology*, 2nd ed. Sage, 2004.

[31] T.-D. B. Le, R. J. Oentaryo, and D. Lo, "Information retrieval and spectrum based bug localization: Better together," in *Proceedings of the Joint Meeting on Foundations of Software Engineering (ESEC/FSE'15)*, 2015, pp. 579–590.

[32] O. A. L. Lemos, A. C. d. Paula, H. Sajnani, and C. V. Lopes, "Can the use of types and query expansion help improve large-scale code search?" in *Proceedings of the International Working Conference on Source Code Analysis and Manipulation (SCAM'15)*, 2015, pp. 41–50.

[33] Z. Li, T. Wang, Y. Zhang, Y. Zhan, and G. Yin, "Query reformulation by leveraging crowd wisdom for scenario-based software search," in *Proceedings of the Asia-Pacific Symposium on Internetware (Internetware'16)*, 2016, pp. 36–44.

[34] X. A. Lu and R. B. Keefer, "Query expansion/reduction and its impact on retrieval effectiveness," *NIST Special Publication*, pp. 231–231, 1995.

[35] F. Lv, H. Zhang, J. g. Lou, S. Wang, D. Zhang, and J. Zhao, "Codehow: Effective code search based on api understanding and extended boolean model (e)," in *Proceedings of the International Conference on Automated Software Engineering (ASE'15)*, 2015, pp. 260–270.

[36] C. D. Manning, M. Surdeanu, J. Bauer, J. R. Finkel, S. Bethard, and D. McClosky, "The stanford corenlp natural language processing toolkit," in *Proceedings of the Annual Meeting of the Association for Computational Linguistics (ACL'14)*, 2014, pp. 55–60.

[37] A. Marcus, A. Sergeyev, V. Rajlich, and J. I. Maletic, "An information retrieval approach to concept location in source code," in *Proceedings of the Working Conference on Reverse Engineering (WCRE'04)*, 2004, pp. 214–223.

[38] A. Marcus and S. Haiduc, "Text retrieval approaches for concept location in source code," in *Software Engineering: International Summer Schools, ISSSE 2009-2011, Salerno, Italy. Revised Tutorial Lectures*, ser. Lecture Notes in Computer Science. Springer, 2013, vol. 7171, pp. 126–158.

[39] C. Mills, G. Bavota, S. Haiduc, R. Oliveto, A. Marcus, and A. De Lucia, "Predicting query quality for applications of text retrieval to software engineering tasks," *Transactions on Software Engineering and Methodology*, vol. 26, no. 1, pp. 3:1–3:45, 2017.

[40] L. Moreno, J. Treadway, A. Marcus, and W. Shen, "On the use of stack traces to improve text retrieval-based bug localization," in *Proceedings of the Conference on Software Maintenance and Evolution (ICSME'14)*, 2014, pp. 151–160.

[41] A. T. Nguyen, T. T. Nguyen, J. Al-Kofahi, H. V. Nguyen, and T. N. Nguyen, "A topic-based approach for narrowing the search space of buggy files from a bug report," in *Proceedings of the International Conference On Automated Software Engineering (ASE'11)*, 2011, pp. 263–272.

[42] B. D. Nichols, "Augmented bug localization using past bug information," in *Proceedings of the Annual Southeast Regional Conference (ACMSE'10)*, 2010, pp. 1–6.

[43] L. Nie, H. Jiang, Z. Ren, Z. Sun, and X. Li, "Query expansion based on crowd knowledge for code search," *IEEE Transactions on Services Computing*, vol. 9, no. 5, pp. 771–783, 2016.

[44] M. F. Porter, "An algorithm for suffix stripping," *Program*, vol. 14, no. 3, pp. 130–137, 1980.

[45] M. M. Rahman and C. K. Roy, "Quickar: Automatic query reformulation for concept location using crowdsourced knowledge," in *Proceedings of the International Conference on Automated Software Engineering (ASE'16)*, 2016, pp. 220–225.

[46] ——, "Strict: Information retrieval based search term identification for concept location," in *Proceeding of the Conference on Software Analysis, Evolution, and Reengineering (SANER'17)*, 2017, pp. 79–90.

[47] S. Rao and A. Kak, "Retrieval from software libraries for bug localization: a comparative study of generic and composite text models," in *Proceedings of the Working Conference on Mining software repositories (MSR'11)*, 2011, pp. 43–52.

[48] M. Roldan-Vega, G. Mallet, E. Hill, and J. A. Fails, "Conquer: A tool for nl-based query refinement and contextualizing code search results." in *Proceedings of the International Conference on Software Maintenance (ICSM'13)*, 2013, pp. 512–515.

[49] R. Saha, M. Lease, S. Khurshid, and D. Perry, "Improving bug localization using structured information retrieval," in *Proceedings of the International Conference on Automated Software Engineering (ASE'13)*, 2013, pp. 345–355.

[50] C. B. Seaman, "Qualitative methods in empirical studies of software engineering," *IEEE Transactions on Software Engineering*, vol. 25, no. 4, pp. 557–572, 1999.

[51] D. Shepherd, Z. P. Fry, E. Hill, L. Pollock, and K. Vijay-Shanker, "Using natural language program analysis to locate and understand action-oriented concerns," in *Proceedings of the International Conference on Aspect-oriented Software Development (AOSD'07)*, 2007, pp. 212–224.

[52] B. Sisman and A. Kak, "Incorporating version histories in information retrieval based bug localization," in *Proceedings of the Working Conference on Mining Software Repositories (MSR'12)*, 2012, pp. 50–59.

[53] B. Sisman and A. C. Kak, "Assisting code search with automatic query reformulation for bug localization," in *Proceedings of the Working Conference on Mining Software Repositories (MSR'13)*, 2013, pp. 309–318.

[54] J. Starke, C. Luce, and J. Sillito, "Searching and skimming: An exploratory study," in *Proceedings of the International Conference on Software Maintenance (ICSM'09)*, 2009, pp. 157–166.

[55] A. J. Viera, J. M. Garrett *et al.*, "Understanding interobserver agreement: The kappa statistic," *Family medicine*, vol. 37, no. 5, pp. 360–363, 2005.

[56] S. Wang, D. Lo, and J. Lawall, "Compositional vector space models for improved bug localization," in *Proceedings of the Conference on Software Maintenance and Evolution (ICSME'14)*, 2014, pp. 171–180.

[57] S. Wang and D. Lo, "Version history, similar report, and structure: Putting them together for improved bug localization," in *Proceedings of the 22nd International Conference on Program Comprehension (ICPC'14)*, 2014, pp. 53–63.

[58] ——, "Amalgam+: Composing rich information sources for accurate bug localization," *Journal of Software: Evolution and Process*, vol. 28, no. 10, pp. 921–942, 2016.

[59] C.-P. Wong, Y. Xiong, H. Zhang, D. Hao, L. Zhang, and H. Mei, "Boosting bug-report-oriented fault localization with segmentation and stack-trace analysis," in *Proceedings of the Conference on Software Maintenance and Evolution (ICSME'14)*, 2014, pp. 181–190.

[60] X. Ye, R. Bunescu, and C. Liu, "Mapping bug reports to relevant files: A ranking model, a fine-grained benchmark, and feature evaluation," *IEEE Transactions on Software Engineering*, vol. 42, no. 4, pp. 379–402, 2016.

[61] X. Ye, H. Shen, X. Ma, R. Bunescu, and C. Liu, "From word embeddings to document similarities for improved information retrieval in software engineering," in *Proceedings of the International Conference on Software Engineering (ICSE'16)*, 2016, pp. 404–415.

[62] K. C. Youm, J. Ahn, and E. Lee, "Improved bug localization based on code change histories and bug reports," *Information and Software Technology*, vol. 82, pp. 177–192, 2017.

[63] J. Zhou, H. Zhang, and D. Lo, "Where should the bugs be fixed? more accurate information retrieval-based bug localization based on bug reports," in *Proceedings of the International Conference on Software Engineering (ICSE'12)*, 2012, pp. 14–24.

[64] T. Zimmermann, R. Premraj, N. Bettenburg, S. Just, A. Schroter, and C. Weiss, "What Makes a Good Bug Report?" *IEEE Transactions on Software Engineering*, vol. 36, no. 5, pp. 618–643, 2010.