A Systematic Literature Review of Automated Query Reformulations in Source Code Search

MOHAMMAD MASUDUR RAHMAN*, Dalhousie University, Canada
CHANCHAL K. ROY, University of Saskatchewan, Canada

Studies show that software maintenance costs up to 80% of the total budget. As a part of maintenance, software developers often resolve critical bugs to ensure the reliability of their software. They might also need to add new features to their software on a regular interval to stay competitive in the market. These bugs and features are reported as change requests (i.e., technical documents written by software users). Developers consult these documents to implement the required changes in the software code. As a part of change implementation, they often choose a few important keywords from a change request as an ad hoc query. Then they execute the query with a code search engine (e.g., Lucene) and attempt to find out the exact locations within the software code that need to be changed. Unfortunately, even the experienced developers often fail to choose the right search queries. As a consequence, the developers often experience difficulties in detecting the appropriate locations within the code and spend the majority of their time in numerous trials and errors. There have been many studies that attempt to support developers in constructing queries by automatically reformulating their ad hoc queries. In this systematic literature review, we carefully select 70 primary studies on query reformulations from 2,970 candidate studies, perform an in-depth qualitative analysis using the Grounded Theory approach, and then answer six important research questions. We analyze the methodologies, evaluation approaches, and challenges of these primary studies, and provide the statistical evidence on the conducted researches during the last 15 years. Our investigation has reported several major findings. First, to date, eight major methodologies (e.g., term weighting, query-term co-occurrence analysis, thesaurus lookup) have been adopted in query reformulation. Second, the existing studies suffer from several major limitations (e.g., lack of generalizability, vocabulary mismatch problem, weak evaluation, the extra burden on the developers) that might prevent their wide adoption. Finally, we discuss several open issues in search query reformulations and suggest multiple future research opportunities.

CCS Concepts: • Software and its engineering → Software maintenance tools; Traceability; Maintaining software; Search-based software engineering; Software reverse engineering.

Additional Key Words and Phrases: Concept location, bug localization, Internet-scale code search, automated query reformulation, term weighting, query quality analysis, machine learning, systematic literature review

ACM Reference Format:
Mohammad Masudur Rahman and Chanchal K. Roy. 2021. A Systematic Literature Review of Automated Query Reformulations in Source Code Search. ACM Comput. Surv. XX, X, Article XX (August 2021), 68 pages. https://doi.org/0000001.0000001

1 INTRODUCTION

Software maintenance costs up to 80% of the total budget in modern software development [45, 52]. One of the major aspects of maintenance is to deal with software bugs and failures that could lead
to massive fatalities (e.g., Boeing-737 MAX crashes [1, 2, 4, 7], Therac-25 accidents\(^1\)). Software developers thus deal with hundreds of bugs and failures to ensure the reliability of their software products [10]. They might also need to frequently enhance the features of their software to stay competitive in the market. We commonly call these maintenance tasks as change tasks for the sake of brevity. Each of these change tasks is triggered by a software change request either in the form of a bug report [169] or simply a feature request containing plain texts [62, 75, 133]. Developers consult these change requests to implement the required changes in the software code.

As a part of the change implementation, developers often choose a few important keywords from a change request as an ad hoc query. Then they execute the query with a code search engine (e.g., Lucene [3]) and attempt to find out the exact locations within the software code that need to be changed as a part of fixing bugs or enhancement of features. Unfortunately, this has been challenging and even the experienced developers often fail to choose the right search queries from a change request [75]. As a result, they experience difficulties in detecting the appropriate locations within the code and spend the majority of their time in numerous trials and errors [75, 82, 96]. One might think of using all the keywords of a change request as a query. However, the change requests are not originally written to be used as search queries. Thus, they often make verbose queries, which are noisy and ineffective [34]. The developers might also attempt to improve their ad hoc queries by adding suitable keywords from the software code. However, Furnas et al. [46] suggest that there is a little chance (e.g., 10%–15%) that they might guess the right keywords from the software code. Besides, the software code has a smaller vocabulary than that of regular texts (e.g., change requests) [60]. In short, selecting the right keywords either from the change requests or from the software code is a major challenge during code search. Thus, the developers are badly in need of automated supports for constructing appropriate search queries and detecting relevant software code during the maintenance.

Developers extensively search for code within a local codebase when dealing with bugs and features. However, they also frequently search for relevant code snippets on the Internet (e.g., GitHub [6]) as a part of fixing bugs or feature enhancement [89]. According to an earlier study [22], the developers spend about 19% of their programming time searching for relevant code on the Internet. During this code search, they also choose ad hoc search queries using a few keywords. However, 76% of their queries need one or more reformulations to succeed [12, 14]. Thus, regardless of the problem context, choosing the right queries and locating the desired code are challenging, which warrants appropriate automated supports for the developers.

Automated support for constructing queries and then for searching the code of interest (e.g., software bugs, features) has been an active area of research for over a decade. A number of studies [31, 34, 47, 56, 58, 62, 65, 75, 133–136, 138, 153, 159, 179] attempt to support developers either (a) by constructing search queries for them from a change request or (b) by reformulating their chosen ad hoc queries. These studies adopt various methodologies and algorithms as follows. First, there have been several studies that construct queries from a change request by employing term weighting algorithms [43, 75, 136, 138, 185], natural language discourse analysis [31] and meaningful heuristics [75]. Second, there have been other studies that reformulate (or reconstruct) a given search query by employing relevance feedback mechanism [47, 56, 58, 101, 135, 159], spatial code proximity analysis [110, 157, 159], query difficulty estimation [54, 56, 57, 135], term-query co-occurrence analysis [62, 134, 181], thesaurus lookup [62, 153] and software repository mining [65, 134, 179, 181]. Third, there have been another group of studies that reformulate queries for Internet-scale code search by capturing complementary keywords from crowdsourced contents [118, 186], well established thesauri (e.g., WordNet) [89, 90, 99] and software repositories [30, 107, 109, 118, 139, 144]. Thus,

\(^1\)https://bit.ly/2KU9IR2
there have been a significant number of studies that reformulate queries to support source code search. Unfortunately, to the best of our knowledge, there exists no systematic literature review that analyzes, categorizes or critically examines these studies, which is essential to advance the state of research on this topic—automated query reformulations for source code search.

Contributions: In this systematic literature review, we investigate the existing researches on automated query reformulations supporting code search that were conducted during the last 15+ years. We carefully select 70 primary studies out of 2,970 candidates that were collected from 11 widely used publication databases (e.g., ACM Digital Library, IEEE Xplore, Fig. 4). Then we critically examine each of these primary studies by going through their full texts and by identifying their methodologies for reformulating queries, evaluations, and limitations. We apply the Grounded Theory approach in our analysis using three levels of coding (open, axial, and selective) [51]. The method has been widely used for literature reviews since it can derive important theories that are firmly grounded within the qualitative data [31, 80, 161]. Using this method, we categorize our selected studies into several categories in terms of their adopted methodologies and reported limitations. We also provide statistical evidence of conducted researches during the last two decades. We also compare and contrast between query reformulations in local code searches and that of Internet-scale code search. Finally, we discuss several open questions in automated query reformulations (supporting code search) and suggest the future research directions in this domain. We thus make the following contributions in this article.

(a) (RQ1) Benchmark. We provide a comprehensive benchmark of 70 primary studies on automated query reformulations supporting code search that were conducted over the last 15+ years (Table 11). They were carefully selected from a total of 2,970 candidate studies through six levels of noise filtration [80, 126] (Fig. 4, Section 3). We present the statistical evidence of conducted researches and discuss the recent trends of this topic (RQ4). We also provide a replication package [8] to reproduce our work.

(b) (RQ1) Taxonomy based on methodology and algorithm. We identify the methodologies or algorithms used by primary studies to reformulate their queries and classify the studies based on these dimensions. We found that, to date, eight major methodologies and algorithms have been used to reformulate queries for code search (Table 2). About 40% of these studies use term weighting and relevance feedback mechanism, 46% make use of semantic relations, word co-occurrences, and thesaurus lookup, whereas 44% of the studies rely on data mining and API recommendations to reformulate their search queries.

(c) (RQ2) Evaluation and validation. We analyze the evaluation methods, performance metrics, and validation targets used by the primary studies and report our findings. About 54% of our primary studies use empirical evaluation, 33% perform qualitative investigation, whereas 9% of the studies use both. They use a total of 30 performance metrics including the popular ones (e.g., Hit@K, MAP, Recall) (Fig. 11). However, human participation is generally low (Fig. 15), and the number of queries used to evaluate queries in the Internet-scale code search is often small (Table 6). About 30% of our primary studies were also selected for comparison by the later studies.

(d) (RQ3) Limitations and challenges of primary studies. We analyze the limitations and challenges of the primary studies that were indicated either explicitly or implicitly in their papers. We found that these studies suffer from eight major limitations or challenges (Table 9) including noisy keywords in their queries, vocabulary mismatch problem, lack of generalizability, human-induced biases, weak evaluation, and other prevailing issues that might prevent them from adoption by the software practitioners.
(e) **(RQ₅)** Similarities and differences between local and Internet-scale code search. About 58% of our 70 primary studies perform query reformulations to support local code searches (e.g., concept location, bug localization, feature location) whereas the remaining 42% focus on Internet-scale code search. The primary studies from these two searches differ in their adopted methodologies, reformulation types, evaluation methods, and developer participation. However, surprisingly, both groups of studies suffer from a common set of issues such as a lack of generalizability, weak evaluation, and noisy queries (Fig. 26). Future work on code search can benefit from these insights during their context-specific tool development.

(f) **(RQ₆)** Open challenges and research opportunities. We identify several open challenges in automated query reformulations supporting code search and determine the scope of future work (Section 5). We found that the state-of-the-art keyword selection algorithms might not be enough to identify appropriate keywords either from change requests or from software code. Future work can focus on adding more contexts (e.g., time-awareness) to the keyword selection algorithms, designing an appropriate fitness function for GA-based solutions [115], leveraging the structures from source code [93, 135], or using neural language modeling (e.g., word embeddings [19, 112, 181]) to deliver better quality queries for the code search.

**Structure of the article:** The rest of this article is organized as follows. Section 2 presents the background concepts and terminologies of automated query reformulations. Section 3 discusses the methodology of our systematic literature review. Section 4 presents our review results where Section 4.1 focuses on the methods, algorithms and data sources used in the primary studies. Section 4.2 discusses the evaluation methods of the primary studies whereas Section 4.3 points out their limitations and challenges. Section 4.5 compares and contrasts between local code searches and Internet-scale code search in terms of their query reformulations. Section 5 discusses the open challenges and scopes for future research in this domain. Section 6 identifies the threats to validity of our findings, Section 7 discusses the related parallel surveys, and finally, Section 8 concludes our article with the major findings.

2 BACKGROUND

2.1 Automated Query Reformulation

Searching for relevant code is a frequent activity in modern software development [22]. One major part of this search operation is to choose an appropriate query that reflects the information need. Unfortunately, during code search, software developers fail even 88% of the time to choose the right search queries [75, 82]. When their search query fails, they attempt to reformulate the query (a) by adding new keywords, (b) by removing noisy keywords, or (c) by replacing the existing keywords with more appropriate ones. When these modifications are performed using automated means, they are called **automated query reformulations** [56, 101, 153]. Thus, the query reformulation tasks are three types as follows:

(a) **Query expansion:** A given search query is expanded with synonyms, semantically similar, or complementary keywords. Developers often need to expand their initial queries since they are generally short (e.g., 2–3 keywords on average) [44, 144]. According to existing studies [14, 144], up to 76% of developer queries need to be expanded during code search.

(b) **Query reduction:** A given search query is reduced by removing the noisy, ambiguous, or less discriminating keywords. The keywords that occur in more than 25% of the documents in a corpus are less discriminating [56, 134]. They are often removed from a query during code search.
User → Change request → Developer

Search query → Code search → Local codebase

Query suggestion → Query reformulation

(c) **Query replacement**: The keywords of a given query are replaced with more appropriate keywords. Examples of query replacement include spelling corrections [48], query generalization, and query specialization [99, 113]. During code search, developers often learn new information from their unsuccessful searches and then calibrate their search by replacing the old query keywords with more appropriate keywords.

### 2.2 Context of Query Reformulation

Context refers to the environment where a task is performed. It has been found to be useful for many software engineering tasks (e.g., code completion, exception handling) [11, 37, 49, 64, 132]. Similarly, the context plays a key role in query reformulation for the code search. In particular, the queries and their reformulation practices often depend on the type of code search they are intended for. Based on the locality and type of a corpus, code searches can be divided into two major categories as follows.

(a) **Local code search**: The code is searched within a local codebase that contains only one software project. Fig. 1 shows how queries are constructed from a change request in the context of local code search. The developer selects either the whole texts or a few important keywords from the change request as an ad hoc query (Steps 1-4, Fig. 1). Then the query is executed by a code search engine that returns the source code that needs to be modified (Steps 5-6, Fig. 1). If the query fails, it is reformulated with automated techniques by carefully analyzing the change request texts and the retrieved source code (Step 7, Fig. 1). Software bugs and features are located within source code using the local code search.

(b) **Internet-scale code search**: The code is searched within an Internet-scale codebase that contains thousands if not millions of software projects (e.g., GitHub, SourceForge). Fig. 2 shows how search queries are reformulated in the context of Internet-scale code search. The developer selects a few keywords as an ad hoc query and then attempts to find out the relevant code from the corpus using a search engine (Steps 1-4, Fig. 2). If the query fails, it is reformulated with automated techniques using thesaurus lookup [99, 139, 186] or intent refinement [68]. Software developers spend about 19% of their programming time searching for relevant, reusable code snippets from the Internet [22].

---

2https://github.com
3https://sourceforge.net/
2.3 Steps of Automated Query Reformulation

During code search, queries are reformulated in multiple steps. These steps might vary based on either type of reformulation or type of code search. However, most of the existing approaches [47, 56, 57, 135, 136, 159] share a common set of steps (e.g., Fig. 3) as follows.

(a) **Query feedback collection:** The first step of query reformulation is to collect feedback on a query (Steps 2–4, Fig. 3). One way to collect the feedback is relevance feedback method [147], which has been widely used in the Information Retrieval domain. In this method, each of the documents retrieved by a given query is examined and annotated as either relevant or irrelevant to the query [47, 101]. The relevance feedback can be of three types as follows.

- **Explicit relevance feedback:** Developers provide explicit feedback and annotate each of the retrieved documents (by a search query) as either relevant or irrelevant [47]. Although the explicit feedback could be accurate and meaningful, capturing it regularly from the developers is time-consuming and sometimes even impossible.
• **Implicit relevance feedback**: Since explicit feedback is costly, several studies capture low-cost feedback that is *implicitly* provided by the developers [73]. This feedback is derived from a developer’s reactions towards the retrieved documents such as eye movements, document examination patterns, and keyword deletion or retention patterns.

• **Pseudo-relevance feedback**: Unlike the above two types, this feedback does not warrant any developer intervention. That is, the Top-K (e.g., K=5) source documents retrieved by a given query are naively considered as *relevant* to the query [56, 62, 147]. It is also known as *blind relevance feedback*. There has been significant evidence [56, 134, 135, 138] that suggests the effectiveness of this relevance feedback mechanism in query reformulation.

(b) **Candidate keyword selection**: Once relevance feedback is collected, the next step is to select appropriate keywords to reformulate the given query using the feedback. Many studies [56, 136, 159] analyze the annotated documents collected as a part of feedback and attempt to find out important keywords from them using various term weighting algorithms (Steps 5–6, Fig. 3). Term weight is a numerical proxy to the relative importance of a keyword within a document [72]. Many of the term weighting algorithms (e.g., Section 2.4) are borrowed from the Information Retrieval domain.

(c) **Reformulation of the search query**: Once the candidate keywords are collected, they are ranked and the top few keywords are used to expand a given query (Step 7, Fig. 3). Less important keywords and the keywords non-existent in the corpus are discarded from the given query [56, 141]. A few studies [56, 114, 135] argue that the same term weighting algorithm might always not work. Thus, they construct multiple reformulated versions of a given query using multiple term weighting algorithms and then deliver the best-reformulated query using supervised machine learning (Steps 8-10, Fig. 3).

2.4 **Term Weighting**

Term weighting is a popular method to determine the relative importance of individual terms within any body of texts (e.g., change request, source code) [72]. It is often used to select candidate keywords during automated query reformulations (Step 6, Fig. 3). Several term weighting algorithms had been proposed and widely used over the last few decades. They can be classified into three broad categories as follows.

(a) **Frequency-based term weighting**: The importance of a keyword is determined based on its occurrence frequencies. The arguably most popular frequency-based method is TF-IDF [72]. TF-IDF stands for Term Frequency (TF) times Inverse Document Frequency (IDF). While TF counts the occurrences of a term (or keyword) within a document, the IDF counts the number of documents (within a corpus) containing the term. Thus, TF-IDF determines a term’s importance without considering the dependencies that it might have on other terms.

(b) **Graph-based term weighting**: Unlike the frequency-based methods, graph-based methods [17, 136, 138] capture the dependencies among keywords during their weight calculation. Existing studies capture co-occurrences [111], syntactic dependencies [17], and hierarchical dependencies [138] among the keywords, and transform a text document into a graph structure. In this graph, the keywords are represented as nodes and their dependencies as edges. Then the important nodes (or keywords) are identified using a graph-based, recursive algorithm namely PageRank [23]. Source code is generally scarce in vocabulary but rich in structures and dependencies [60]. Thus, graph-based algorithms leveraging these dependencies could be a suitable choice to select important keywords from the source code during query reformulations [135, 138].
Probabilistic term weighting: Unlike the above two types, probabilistic term weighting methods make use of Information Theory to calculate the term weights [24, 26, 28]. One of the popular probabilistic methods is Kullback–Leibler Divergence (KLD) that determines the divergence of a keyword (within a document) from the random occurrence probability. That is, if a keyword is frequent across numerous documents within a corpus (a.k.a., generic keyword), it does not contain unique information, which makes it less important. On the contrary, a keyword that is specific to only a few documents often contains meaningful information, which makes it important.

2.5 Implications of Automated Query Reformulation
Choosing the right queries during code search is a challenging problem. Hence, automated supports for constructing queries are highly warranted for the developers. However, automatic reformulations of queries have both positive and negative implications as follows:

(a) Benefits of query reformulation: Developers often use short queries to search for relevant code, which might not always reflect their information need [44]. Thus, expanding these short queries with similar or complementary keywords often improves their performance. According to an existing study [100], up to 20% performance gain could be achieved in the search using automatically reformulated queries. Carefully reformulated queries can also help the developers locate their desired code with reduced cognitive or manual efforts.

(b) Costs of query reformulation: Automatic reformulations might hurt the search queries that are already good (or appropriate) [27, 56, 135]. Adding extra keywords makes them noisy [155]. Thus, no reformulation is better for them than inappropriate reformulations [121, 155, 160]. There also exist a few difficult queries that cannot be improved using the traditional reformulation approaches [27, 50].

Given these two-fold implications of query reformulations, contemporary approaches attempt to maximize the benefits and minimize the costs of reformulation. For example, several studies [56, 57, 135] adopt supervised machine learning and query quality analysis to improve the poor queries and to preserve the good queries during the code search.

3 METHODOLOGY
We use systematic literature review (SLR) [80, 81] to conduct our survey, as was used by several earlier studies [126, 128, 161]. We start our investigation by asking six research questions, and then provide a comprehensive analysis of the topic – automated query reformulations in code search. We provide both quantitative and qualitative evidence of the relevant research, and establish the state-of-the-art practices. Our goals are to summarize the current state of research in this domain and to outline the future research directions. We refer to the guidelines of Kitchenham and Charters for our survey as follows:

3.1 Research Questions
Specifying the research questions is one of the most important steps of the systematic literature review [80, 81]. We ask six research questions in our survey. Our questions are classified into three categories – general questions, statistical questions and focused questions. Table 1 shows our six research questions.

Our general questions target the generic aspects of query reformulation approaches such as their algorithms, methodologies, or used corpora (RQ1), their evaluation and validation methods (RQ2) and the challenges or limitations they experience (RQ3). Our statistical questions attempt to gather the statistical evidence of the conducted researches over the years (RQ4). Finally, our
focused questions concentrate on the specific aspects of automated query reformulations targeting code search. They compare between local code search and Internet-scale code search in terms of their query reformulation practices (RQ5), develop the guidelines for designing effective query reformulation tools, and also identify the future research directions (RQ6).

3.2 Search Strategy

The next step of the systematic literature review is to collect a complete set of primary studies that can help us answer the research questions. This step involves the construction of appropriate search keywords on a research topic and then the accumulation of a non-biased set of studies from appropriate publication databases (e.g., IEEE Xplore, ACM Digital Library).

Construction of search keywords: Choosing appropriate keywords and optimizing them for search are crucial to the discovery of available studies on a research topic. Research questions determine the scopes and motivations of a systematic review. Kitchenham and Charters suggest that these questions should be broken down into individual facets such as search keywords. Then they should be expanded with their synonyms, abbreviations, and alternative spelling using boolean operators. Earlier studies often use a PIO (Population + Intervention + Outcome) criterion [80, 123] to identify their search keywords. The PIO approach determines three types of keywords as follows:

Population terms encompass all the aspects of a research topic including its application scopes, technologies, and standards. In the context of automated query reformulation targeting code search, we thus consider the following terms as the population of search keywords.

| Information retrieval, IR, text retrieval, TR, bug localization, concept location, feature location, FLT, concern location, Internet-scale code search, code search engine, search engine, local code search, code search, source code search, and code search query. |

Intervention terms focus on the specific aspect of a topic under study. In the context of automated query reformulation, we thus consider the following terms as our intervention keywords.

| Query formulation, query expansion, query reduction, query formulation, query refinement, automated query expansion, AQE, query suggestion, query recommendation, term selection, query replacement, query difficulty, query quality, keyword selection, keyword extraction, search term identification, search query, search term, and search keyword. |

Outcome terms are related to the factors that are of importance to the stakeholders (e.g., developers). In the context of query reformulation targeting code search, these outcomes could be the improvement in result ranks, reduced effort in code search, and reduced effort in localizing the bugs. However, we did not want to restrict our search too much. Thus, we do not include the outcome terms in our search keywords like the earlier study [80]. Finally, we construct our keyword set for the systematic literature review as: Population AND Intervention.

Source of information: Our goal was to collect as many as studies possible on the topic of interest from the literature. We choose 11 electronic publication databases for our study. These databases store most of the researches conducted in Computer Science and Software Engineering [80]. In particular, we collect all the journal articles and conference papers published between 1998 and 2021. We choose 1998 as the starting year since it is the inception year of Google’s PageRank, the algorithm that has revolutionized web search. Our investigation finishes with 2021 as the ending year. First, 2,871 candidate studies, conducted between 1998 and 2018, were collected as a part of the PhD comprehensive exam of the first author. Recently, we include 99 more studies conducted between 2018 and 2021. We thus collect a total of 2,970 potentially relevant articles and papers on query reformulations targeting code search for our survey.
Table 1. Research Questions

| Ref# | Question                                                                 |
|------|--------------------------------------------------------------------------|
|     | **General questions**                                                    |
| **RQ**1 | Which methods, algorithms and data sources have been used for automated query reformulations targeting code search in the literature? |
| **RQ**2 | Which methods, metrics or subject systems have been used to evaluate and validate the researches on automated query reformulations? |
| **RQ**3 | What are the major challenges of automated query reformulations intended for code search? How many of them have been solved to date by the literature? |
|     | **Statistical questions**                                                |
| **RQ**4 | How much activities of research on automated query reformulations have been performed to date? What are the venues that these researches got published at? |
|     | **Focused questions**                                                   |
| **RQ**5 | What are the differences and similarities between query reformulations for local code search and query reformulations for Internet-scale code search? |
| **RQ**6 | What are the scopes for future work in the area of automated query reformulation targeting the code search? |

| Source                  | Initial Search | Impurity Removal | Filtered by Title (C1&C2) | Filtered by Abstract (C1&C2) | Combined (2004–2018) | Duplicate Removal | Filter by Full Texts (C3) | Final Selection |
|-------------------------|----------------|------------------|---------------------------|-----------------------------|----------------------|------------------|--------------------------|-----------------|
| ACM Digital Library     | 176            | 114              | 40                        | 16                          |                      |                  |                          |                 |
| CrossRef                | 152            | 101              | 27                        | 7                           |                      |                  |                          |                 |
| DBLP                    | 216            | 179              | 74                        | 10                          |                      |                  |                          |                 |
| Mendeley                | 203            | 123              | 59                        | 30                          |                      |                  |                          |                 |
| Google Scholar          | 200            | 173              | 81                        | 18                          |                      |                  |                          |                 |
| IEEE Explore            | 432            | 299              | 65                        | 41                          |                      |                  |                          |                 |
| ProQuest                | 137            | 127              | 68                        | 38                          |                      |                  |                          |                 |
| ScienceDirect           | 22             | 14               | 04                        | 04                          |                      |                  |                          |                 |
| SpringerLink            | 313            | 265              | 47                        | 03                          |                      |                  |                          |                 |
| Web of Science (ISI)    | 999            | 912              | 91                        | 26                          |                      |                  |                          |                 |
| Wiley Online Library    | 21             | 10               | 04                        | 02                          |                      |                  |                          |                 |

**Total:** 2,871 2,317 562 195 109 93 62 56

(2018–2021)

| Source                  | Final Selection |
|-------------------------|-----------------|
| Google Scholar          | 99              |

Fig. 4. Selection of primary studies
3.3 Study Selection

We exclude irrelevant studies from the 2,970 initial results, and select only the relevant ones for our systematic review. We apply three different filters, and choose the ones as our primary studies that meet certain conditions and quality standards. In the context of our systematic literature review, we carefully exclude such studies that:

- $C_1$: do not address code search
- $C_2$: do not address query reformulation
- $C_3$: do not address query reformulation for code search

Fig. 4 shows different steps of our study selection process. We apply multiple filters on the initial search results across several steps, and choose the final set as follows:

(a) Removal of impurities: Initial search results often contain entries that are not related to software engineering. This might happen due to accidental keyword matching between the query and the full texts from the papers. We use a semi-automatic approach to remove these impurities and to ensure the quality of our results. First, we analyze the keywords from each of the result titles and carefully identify the words that are totally at odd with Software Engineering and code search. Then we discard the studies that contain any of these keywords.

Image, multimedia, multilingual, bilingual, video, library, libraries, digital, database, databases, db, chinese, japanese, trec, mobile, medical, geo, music, sql, audio, and speech

According to our investigation, query reformulations have been adopted in the application domains above, which are clearly not related to source code search. We thus removed 554 (18.65%) impure results, and our initial studies got reduced to 2,416 (81.35%) after this step.

(b) Noise filtration using title: After the removal of impurities, we are still left with thousands of results. Manually analyzing them for relevance is still impractical. We thus adopt another heuristic to reduce the noise from our results. Since we are interested about the studies related to query reformulation and code search, we discard any studies that do not contain at least one of the following important keywords in their titles.

query, code, developer, change, bug, concept, feature, concern, and software

We believe that the studies that miss these keywords in their titles might not be relevant to our survey. This semi-automated filtration step discarded 1,769 (59.56%) irrelevant studies, and our collection got reduced to 647 (21.78%) potentially relevant studies.

(c) Noise filtration using abstract: Noise filtration using title might not be sufficient enough, and the collection still could contain irrelevant studies. We thus manually analyze the titles from 647 studies and their corresponding abstracts in order to understand the underlying research. We noticed that despite having the desired keywords, many of these studies do not deal with query reformulations and code search. We thus manually discarded 422 (14.21%) studies, and our collection got reduced to 225 (7.58%) relevant studies.

(d) Result merging & duplicate removal: In this step, we merge the relevant studies from each of the 11 publication databases, and discard the duplicate studies. Since the same study could be indexed by multiple databases, the duplication might occur during retrieval. Even multiple versions of the same study (e.g., peer-reviewed, pre-print) could be retrieved from multiple databases. We thus manually analyzed 225 studies and discarded a total of 105 (3.54%) duplicate studies. This step left us with a total of 120 (93 + 27) relevant studies.

(e) Noise filtration using full texts: We were unable to determine the subject matter of several studies even after reading their titles and abstracts. We thus read the full texts of 120 research papers, and discard several irrelevant studies. Many of these studies discuss either code search...
algorithms or query reformulations in general. Since our topic was query reformulation for code search, we considered them irrelevant for our survey and discarded them from the collection. This filtration left us with 76 studies. We also discarded short papers (e.g., poster, doctoral symposium) and non peer-reviewed items (e.g., technical reports, dissertation). Since our focus was only on the techniques for query reformulations, we also discarded such studies that conducted developer surveys, interviews, or case studies on concept location, feature location and Internet-scale code search. Then we added a few studies that were relevant (according to our prior work experience) but missed by the search process accidentally. Finally, we chose a collection of 70 (2.36%) papers as the primary studies for our systematic review.

3.4 Quality Assessment

Table 14 (Appendix A) provides an overview of our selected primary studies, their adopted methodologies, and offered features. As a standard practice, we investigate whether these studies meet certain quality criteria or not [80, 81]. In particular, we carefully go through each of these 70 studies, capture their subject matters, and then answer 10 standard quality questions as follows:

- Q1: Is there a clear statement about the aim of the research?
- Q2: Is there adequate description about the context of the research?
- Q3: Is there a review about the related work from the literature?
- Q4: Is there adequate description on a query reformulation technique targeting code search (e.g., bug localization, concept location, Internet-scale code search)?
- Q5: Has the approach been validated?
- Q6: Does the conclusion reflect the aim or purpose of the conducted study?
- Q7: Is there a clear statement of findings?
- Q8: Does the study discuss its limitations or threats to validity?
- Q9: Does the study recommend further research?
- Q10: Does the study provide any supports for the replication (e.g., replication package)?

Table 15 (Appendix B) shows responses to each of the questions above. We record the response as either “Yes” (●), “No” (○) or “Somewhat Yes” (◇, ●). We record “Yes” when the supporting evidence is clearly visible, “No” when the supporting materials are not found at all, and “Somewhat Yes” when the relevant evidence is weak or ambiguous. From Table 15, we see that majority of our selected studies meet the quality criteria. While a few studies do not provide an exhaustive description of their query reformulation approach, they are still retained in this survey due to their overall relevance. Many of these studies also do not provide a publicly available replication package. However, it does not hurt the overall goal of our literature review. Thus, they were also retained.

3.5 Grounded Theory Based Analysis

We use Grounded Theory approach [51] to analyze our primary studies. It is widely used in the social science researches to derive meaningful theories that are firmly grounded on the data (e.g., interview scripts). Recently, this approach has also found applications in the Software Engineering researches [31, 129, 152]. We systematically analyze the full texts from each of the primary studies, derive important theories using this approach, and then answer several of our research questions. Grounded theory approach involves three stages of coding as follows:

(a) Open coding involves breaking down the gathered data into identifiable, interesting chunks and annotating them with appropriate key phrases (a.k.a., open codes) [129]. We analyze the full text of each primary study, attempt to understand a concept of interest (e.g., methodology, validation, shortcomings), and then record our observations using suitable key phrases. The key idea was to keep an open mind and choose as many as codes needed to represent each study. For example,
Table 2. Methodologies used for query reformulation

| ID | Methodology |
|----|-------------|
| M1 | Term weighting and relevance feedback |
| M2 | Dependency graph mining |
| M3 | Semantic relations, co-occurrences, and thesaurus |
| M4 | Domain ontology and syntactic relations |
| M5 | Machine learning, query difficulty analysis, and word embeddings |
| M6 | Data mining and API recommendation |
| M7 | Genetic algorithms |
| M8 | Miscellaneous |

we used 209 open codes [8] to represent the methodologies adopted by our 70 primary studies to reformulate their queries. We spent ≈100 man-hours in this open coding task.

(b) Axial coding establishes relationships among the open codes. In this stage, we place our open codes into a spreadsheet, and establish connections among them using various colours. We annotate the key phrases with the same colour if they are somehow related. We consider not only lexical overlap but also the semantic relatedness to connect the key phrases [129]. Our goal was to convert the open codes into low-level categories. This stage provided a set of 21 tentative categories [8] that explain the methodologies adopted by our 70 primary studies.

(c) Selective coding identifies the core variables behind a given phenomenon [51, 152]. In our case, automated reformulation of a given query is the target phenomenon. While the axial coding provides a set of low-level categories, we carefully analyze them and merge them into high-level categories based on their themes and semantic relatedness. This stage provided eight key categories of methodologies (Table 2) that were used to reformulate search queries by our primary studies. Each of these categories is represented using a set of semantically related key phrases [8].

4 RESULTS

We present our detailed analyses and findings in this section. We carefully go through each of the 70 primary studies, consult their major sections (e.g., problem descriptions, methodologies, algorithms), and then summarize our non-trivial observations using Grounded Theory approach [51]. We divide our findings and discussions into multiple logical sections, and attempt to answer each of our research questions (Table 1) as follows:

4.1 Answering RQ1: Methodologies, algorithms, and data sources used for automated query reformulations

We identify eight key methodologies used by our primary studies for query reformulation. We use the Grounded Theory, a popular qualitative analysis approach, to come up with them. Tables 2, 3 show the methodologies and the primary studies that adopt them. We discuss each of these methodologies, their reported strengths and weaknesses as follows.

(M1) Term weighting and relevance feedback: Term weighting determines the relative importance of a term (or word) within a body of texts (e.g., document). It has been an integral part of Information Retrieval domain [72, 149]. Jones [72] first introduced a term weighting method namely TF-IDF for Vector Space Model (VSM)-based document retrieval. According to Vector Space Model, a text document can be modeled as a vector of distinct words that are found in the document. On the other hand, TF-IDF stands for Term Frequency (TF) × Inverse Document Frequency (IDF) (check Section 2.4 for details). That is, the terms that are frequently used within a target document but rarely used across the whole corpus, are considered to be important within the target document.
Table 3. Primary studies and their adopted methodologies

| ID | Primary studies | Total |
|----|----------------|-------|
| M1 | S1, S2, S4, S7, S9, S10, S11, S12, S14, S19, S21, S22, S24, S25, S26, S30, S34, S38, S40, S43, S45, S50, S51, S54, S59, S61, S64, S70 | 28 (40%) |
| M2 | S4, S19, S22, S24, S25, S31, S33, S40, S46, S56, S57, S61 | 12 (17%) |
| M3 | S2, S3, S4, S6, S13, S15, S16, S17, S18, S19, S20, S21, S23, S24, S27, S28, S35, S37, S39, S40, S41, S42, S43, S45, S49, S50, S57, S58, S60, S65, S70 | 32 (46%) |
| M4 | S1, S5, S12, S16, S17, S18, S36, S37, S42, S43, S47, S48, S49, S51, S53, S62 | 16 (23%) |
| M5 | S8, S10, S11, S22, S29, S32, S35, S58, S62, S63, S66, S67 | 12 (17%) |
| M6 | S2, S7, S12, S15, S16, S17, S18, S19, S20, S26, S28, S34, S35, S37, S38, S39, S41, S43, S46, S49, S51, S52, S54, S55, S57, S58, S61, S63, S65, S66, S70 | 31 (44%) |
| M7 | S9, S62, S67, S68, S69 | 5 (7%) |
| M8 | S1, S3, S4, S5, S6, S7, S11, S12, S13, S14, S15, S17, S21, S24, S26, S29, S37, S39, S41, S44, S46, S53, S57, S60, S62, S67 | 26 (37%) |

Existing studies extract these terms as important keywords from a document [105]. Besides TF-IDF, its variants (e.g., TF, IDF) are also used for term weighting on an ad hoc basis.

A number of primary studies [38, 47, 57, 75, 93, 114, 121, 137, 156, 170, 185] use term weighting methods (e.g., TF-IDF) to reformulate their search queries. In particular, they use these methods in association with relevance feedback mechanism [147]. Relevance feedback is a way to capture complementary information on a query that can be used to reformulate the query (check Section 2.3 for details). First, a set of tentatively relevant documents are collected using the feedback mechanism. Second, candidate terms are extracted from these documents using one or more term weighting methods. Third, the candidate terms are then ranked based on their relative importance (a.k.a., term weights) and only Top-K terms are used to reformulate a query. Hayes et al. [59] first introduce relevance feedback and term weighting (e.g., TF-IDF) in the context of software engineering problems. They reformulate search queries to recover traceability links and to reduce false-positive trace links. Lucia et al. [101] critically examine and extend their work by conducting multiple case studies and using various software artifacts (e.g., use cases, requirements, test cases). However, Gay et al. [47] were the first to use the term weighting and relevance feedback to reformulate search queries in the context of code search (e.g., concept location). Since then many studies adopt the term weighting and relevance feedback in their query reformulation approaches [56, 135, 156, 159, 170]. Haiduc et al. [56] employ Rocchio’s expansion [141], Dice similarity [29], and Robertson Selection Value (RSV) [140] to reformulate a query where they use TF-IDF as a proxy of term importance. Their query difficulty models also make use of several variants of TF-IDF (e.g., avgIDF, maxIDF, maxIDF) [57, 114]. Perez et al. [121] adopt similar query reformulation techniques and term weighting methods to find relevant features from software model corpus. They later combine expert feedback with Rocchio’s method to expand their search queries [122, 124]. Eddy et al. [43] assign varying weights to the terms from various locations of a source document to identify the target software features. Wang et al. [170] use explicit feedback from developers to expand their queries (i.e., Rocchio’s expansion) in the context of Internet-scale code search. However, they use their expanded query to re-rank the results retrieved by a given query rather than the code search.

To reformulate a search query, several studies capture relevant API classes, tags, and software-specific terminologies from a programming Q&A site, Stack Overflow, where they employ various keyword selection methods including term weighting [91, 107, 118, 134, 137]. Rahman and Roy [137] collect relevant Q&A threads from Stack Overflow for a given query using pseudo-relevance feedback, identify the important API classes from their embedded code segments using TF-IDF, and
then use the top-ranked API classes to expand the query. Nie et al. [118] employ Rocchio’s expansion method, and identify software-specific keywords from the Q & A threads of Stack Overflow using TF-IDF. Martie et al. [107] select frequent API classes from the relevant source documents retrieved by a query to expand their query. Kevic and Fritz [75] use TF-IDF and three other lightweight heuristics (e.g., part of speech, position) to identify search keywords from a change request to support the concept location task. Zamani et al. [185] extend the classical term frequency metric (i.e., TF) with time-awareness and improve the feature location task. They suggest that important terms within a source document are not only more frequent but also more recent than the noisy terms. They make use of version control history of a software system to gather the term frequency and timing related meta data. Gibiec et al. [50] make use of relevant web pages from three search engines (e.g., Google, Bing and Yahoo!) to expand the stubborn trace queries. In particular, they extract complementary key phrases from these web pages based on their occurrences (e.g., domain term frequency) and expand the trace queries.

Although TF-IDF has been a popular method of term weighting, it suffers from a major limitation. TF-IDF and other frequency based methods fail to capture the dependencies among terms, which can be important to determine their relative importance [17, 111]. These dependencies could be statistical, syntactic, semantic or hierarchical (check Section 2.4). Many existing studies from Information Retrieval domain [17, 111] make use of these dependencies to determine term weights. However, Rahman and Roy [136] were the first to leverage both statistical and syntactic relationships among terms to formulate queries for code search. They first transform a change request into a text graph using these dependencies and then apply PageRank algorithm to the graph to identify the top search keywords for concept location. They later extend and adapt their approach to source code [135], stack traces [138] and reformulate search queries for IR-based bug localization. They also report significant benefit of using graph-based term weighting over traditional alternatives (e.g., TF-IDF) in query reformulation. Several later studies [77, 78, 125] also adopt graph-based approaches to reformulate their code search queries.

About 40% (28/70) of the primary studies use relevance feedback and term weighting to reformulate their queries during various code search operations (e.g., concept location, bug localization, Internet-scale code search). While relevance feedback captures complementary items on a query (e.g., relevant documents), the term weighting methods identify important keywords from them to reformulate the query. Existing studies leverage several proxies (e.g., frequency, dependencies) to determine the relative importance of search keywords. Thus, the effectiveness of their reformulated queries depend on the appropriateness of these adopted proxies for term importance.

(M2) Dependency graph mining: Several primary studies [15, 53, 138, 170, 171] leverage dependencies among items (e.g., words, API classes) to construct queries for code search. Wang et al. [171] mine an appropriate sub-graph from the program dependency graph (PDG) of a given code segment and construct a dependency query to find structurally similar code segments from a codebase. Their goal was to support code-level changes by automatically detecting the similar code segments. They later incorporate explicit feedback from developers in their code search [170]. Panchenko et al. [119] extract abstract syntax tree (AST) from a given code example and reformulate it into an XPath query to detect structurally similar code examples. Balachandran [15] also address the same problem by extracting appropriate sub-trees from the AST as a reformulated query. Designing search query from a given code segment is often called query by example in the literature [15, 119]. Zhang et al. [188] expand the object-oriented methods (i.e., corpus documents) with neighbouring methods where the neighbours are detected based on their call dependencies, temporal proximity, and semantic similarity. Wang et al. [166] detect syntactic and semantic facets from the results retrieved by a given query and use them to reformulate the query for concept search.
Rahman and Roy also mine three types of graphs – trace graphs, text graphs, and API co-occurrence graphs – to reformulate queries for various code search operations (e.g., bug localization, concept location, Internet-scale code search). Several later studies extend their work where they use graph-based mining to reformulate their queries.

About 16% (11/70) of the primary studies mine dependency graphs to reformulate queries for their code search. They extract dependencies from both structured (e.g., source code) and unstructured items (e.g., texts) to develop these graphs. Unlike texts, the graphs have the potential to capture more contextual information (e.g., call dependencies, AST, syntactic dependencies), which could significantly benefit the reformulated queries.

(M3) Semantic relations, co-occurrences, and thesaurus: Unlike the term weighting and graph-mining studies above, a number of primary studies leverage co-occurrences between query keywords and candidate terms to reformulate their search queries. Sisman and Kak first use spatial code proximity and suggest that terms co-occurring within close proximity in source code are suitable candidates for query expansion. They were the first to expand search queries for IR-based bug localization. They extend their idea later with term ordering and language modeling. Yang and Tan mine the mapping between code and corresponding comment, and construct a software-specific thesaurus – SWordNet. Then they reformulate a given query with semantically similar, alternative words collected from their thesaurus to support the concept location task. Similarly, Howard et al. analyze method signatures and their corresponding leading comments from source code, and extract semantically similar word pairs to support query reformulations. Both approaches above are subject to the availability of sufficient comments or documentations in the source code.

Shepherd et al. extract Verb-Direct Object (V-DO) pairs from method signatures and code comments, and recommend equivalent verbs and DO to support the developers in their query reformulation. They suggest that if two V-DO pairs share the same direct object, then their verbs could be considered as equivalent. However, all free-form queries cannot be modeled as a V-DO pair. Hill et al. extend this work by relaxing the constraints of V-DO pairs and by suggesting phrasal representations of the method signatures. Developers thus could locate query keywords within these phrases, determine the relevance of corresponding methods, and then choose the candidate terms for query expansion. Marcus et al. construct a latent semantic space using Latent Semantic Indexing (LSI) and suggest semantically similar words to reformulate a given query. They also consider the co-occurrences between query keywords and candidate terms within source code to estimate the semantic relatedness.

Several other studies leverage the co-occurrences among items (e.g., terms, tags) within programming Q&A threads and search logs for query reformulation. Stack Overflow, a popular programming Q&A site, discusses software-specific concepts and terminologies in millions of questions and answers. These questions and answers have been frequently used to support several Software Engineering tasks. Li et al. leverage explicit and implicit co-occurrences of tags from the same question and the duplicate questions respectively, and reformulate a given query with semantically similar tags. They calculate Mutual Information (MI) between any two tags with the query keyword being one of them, and then improve a given query using alternative tags. Rahman and Roy analyze the contextual words of both query keywords and candidate terms in the titles of Stack Overflow questions, and suggest appropriate terms for query reformulation based on their contextual similarity. Satter and Sakib mine co-occurrences between query keywords and candidate terms in the past queries from code search logs, and suggest the frequently co-occurred terms for query expansion.
Although co-occurrence between query keywords and candidate terms has been a popular idea for query reformulation, two similar words might always not co-occur in the same context [70, 87]. According to Hussain and Bisson [70], high order co-occurrences could be a possible solution to this issue. However, most of the primary studies above consider only the first order co-occurrences among terms to estimate their semantic relatedness, which might not be enough.

Several primary studies [48, 89, 90, 92, 99, 153, 165] adopt well-established thesauri such as WordNet [113, 172] to reformulate their search queries. Shepherd et al. [153] accept a Verb-Direct Object (V-DO) pair as a search query, and suggest synonyms from WordNet to expand their query. Developers are then responsible to choose the right keywords for their queries during concept location. Ge et al. [48] later extend this work with multi-level keyword suggestion and typo correction leveraging WordNet. Lu et al. [99] replace each keyword of a given query with synonyms extracted from WordNet, and outperform the technique of Hill et al. [62] in concept location task. Lemos et al. [89] also capture synonyms and antonyms from WordNet and predefined types from Java type thesaurus to reformulate queries during Internet-scale code search.

Most of the studies above are inspired by parallel researches on automated query expansion from the Information Retrieval (IR) domain [29]. However, existing findings [160] also suggest that the same word can hold two different semantics for regular texts and source code. For example, the term new represents an adjective in the regular texts whereas it means a verb (e.g., object creation) in the object-oriented source code [179, 180]. In other words, query expansion approaches solely based on English language thesaurus (e.g., WordNet) might not be effective for code search. Given such a limitation, several primary studies [65, 92, 179, 180] focus on developing software-specific thesauri to support query reformulations where they leverage the co-occurrences among keywords within the same context (e.g., method signatures, question tags).

Yang and Tan [180] first construct a software-specific thesaurus namely SWWordNet containing 8.5 million word pairs extracted from nine open source projects. This thesaurus was later used to collect semantically similar words and to reformulate queries for code search [90, 180]. Kevic and Fritz [74] mine change requests, corresponding changed code, and developer’s interactions with the IDE during code-level changes, and construct a dictionary that maps the problem domain vocabulary to solution domain vocabulary. Then they translate a search query (a.k.a., change request) into corresponding source code elements (e.g., methods, classes) using the dictionary to improve the concept location task. Vinayakarao et al. [165] mine programming concepts (e.g., integer array) and corresponding syntactic representations (e.g., int arr[]) from the Q & A threads of Stack Overflow, and construct a mapping database. Then they augment the source code statements with appropriate programming concepts from the database to improve code comprehension tasks. Although thesaurus-based approaches are popular and widely used, they might suffer from the lack of generalizability. That is, the thesauri should be constructed from appropriate corpora (e.g., codebase, Stack Overflow) and must be updated frequently to remain relevant.

About 46% (32/70) of our primary studies make use of term co-occurrences and well-established thesauri (e.g., WordNet, SWWordNet) to reformulate their queries during code search. However, semantically similar terms might always not co-occur in the same context, which warrants for higher order co-occurrences. Thesauri-based approaches might suffer from the lack of generalizability and the thesauri might also require frequent updates to remain relevant.

(M4) Domain ontology and syntactic relations: Ontology defines inter-relations among various concepts of an application domain (e.g., Software Engineering) [174]. Modern software systems are inherently complex and deal with numerous concepts ranging from high level problem descriptions to low level programming solutions. One way to preserve and manipulate this conceptual

ACM Comput. Surv., Vol. XX, No. X, Article XX. Publication date: August 2021.
knowledge is to construct an ontology from these systems. A few of the primary studies [42, 71, 178] leverage the ontology to reformulate their queries during code search.

Dourdas et al. [42] reformulate a problem domain query into a solution service query using their ontology constructed from the source code. They capture the mapping between proven solution classes and requirement classes in their ontology. Wursch et al. [178] construct another ontology using semantic web technology where they capture structural and data dependency relationships among classes, methods and attributes from a software system. Their approach accepts a quasi-natural language query from the developer, reformulates it into a structured SPARQL query, and then retrieves a list of relevant program elements. Unlike Wursch et al., Jiang et al. [71] construct ontological models from multiple software repositories such as source code, bug-fixing history, and bug reports. Then they reformulate a free-form query into multiple candidate questions by capturing the query conditions and then mapping them to ontological concepts. Developers then choose the most appropriate question from them and locate the relevant program elements. Several other studies [50, 93] also make use of domain-specific concepts to reformulate their queries and to improve their search. Although ontology-based studies are reportedly useful to reformulate natural language queries into complex structured queries [178], they are still limited by their underlying technologies (e.g., semantic web, SPARQL). Besides, maintaining an ontology is costly since the changes in source code are frequent and they need to be reflected in the ontology.

Besides conceptual relations (e.g., ontology), a number of primary studies [62, 78, 99, 136, 165, 183, 185] leverage the syntactic or linguistic properties of texts to reformulate their search queries. As a standard practice, they often perform parts of speech (POS) tagging and then select or emphasize the nouns and verbs in their query reformulation. The underlying idea is that these words convey more important semantics than the others. However, POS tagging is often noisy with technical texts (e.g., bug reports, change requests), which could hurt the query reformulation.

About 23% (16/70) of our primary studies make use of conceptual (e.g., ontology) and syntactic relations (e.g., POS tagging) to reformulate their queries during code search. Although they have been used by several studies, maintaining a software-specific ontology could be challenging and the POS tagging with technical texts could be noisy.

(M5) Machine learning, query difficulty analysis, and word embeddings: Several primary studies [25, 56, 57, 67, 75, 97, 114, 135, 177] adopt machine learning to reformulate their queries during code search. Kevic and Fritz [75] capture part of speech, term weight, position and notation of each term from a change request, and train a Logistic Regression model. Their model then accepts an incoming change request and returns Top-3 terms as a search query for concept location. Liu et al. [97] train an encoder-decoder model using deep neural networks and translate a natural language query into relevant API methods to support Internet-scale code search. Wu and Yang [177] suggest that the changed version of code might hold the true intent of a search query that retrieves the original code. Being motivated by this idea, Huang et al. [67] train a deep belief network (DBN) with original code and their changed versions from thousands of GitHub projects, and then expand a given query with appropriate terms extracted from the changed code. Lawrie and Binkley [86] develop an automated summarizer using sequence-to-sequence model and reformulate a bug report into a summary to support the bug localization task. Cao et al. [25] train a sequence-to-sequence model with an attention mechanism where they use original and reformulated queries from the search logs of Stack Overflow Q&A site. Then they use their model to reformulate a given query during code example search at Stack Overflow. Although the sequence-to-sequence models have been popular for reformulating queries, they might require a large amount of training data.

Determining quality of a search query before its actual reformulation has been a popular idea in the Information Retrieval domain [27]. Haiduc et al. [57] first analyze and predict the difficulty of
queries in the context of code search. They train their model using 21 pre-retrieval metrics from the IR domain, and then predict the quality of a given query during concept location. Mills et al. [114] later extend this model with seven post-retrieval metrics (i.e., 28 in total), and evaluate the model performance extensively. Given a search query, Haiduc et al. [56] generate a list of four reformulation candidates using Rocchio’s expansion [141], Robertson Selection Value (RSV) [140], Dice similarity [29] and query reduction [29]. Then they train a query difficulty model to suggest the best reformulated query. Rahman and Roy [135] also extract four reformulation candidates by analyzing method signatures and field signatures from source code and then suggest the best candidate using the query difficulty analysis [57]. Although the query difficulty analysis has been popular and cheap, its metrics might not be as accurate as the query performance metrics (e.g., mean average precision, query effectiveness).

Over the last few years, word embedding technology has been leveraged to tackle the vocabulary mismatch problem in code search. Zhang et al. [186] learn the embeddings of query keywords and API classes from a corpus of ≈25,000 open source projects using a three-layer neural network and Continuous Bag of Words (CBOW) algorithm. Then they translate a natural language query into relevant API classes to support code search. Similarly, Rahman and Roy [137] learn the word embeddings from 1.40 million Q&A threads of Stack Overflow using Skip-gram algorithm [32], and expand a natural language query with relevant API classes. Both studies above report significant performance improvement in their code search.

About 17% (12/70) of our primary studies use machine learning, query difficulty analysis, and word embeddings to reformulate their queries during code search. While these technologies have high potential to improve query reformulations and code search, they are also restricted by their training datasets, learning algorithms, and model parameters.

(M6) Data mining and API recommendation: Many primary studies [25, 93, 102, 127, 137, 139, 150, 156, 186] mine code search logs and software repositories (e.g., API documentation, programming Q&A website) to reformulate their queries. Bajracharya and Lopes [14] first capture the search logs from an Internet-scale code search engine—Koders, and analyze the patterns, topics, and user behaviours during code search from the logs. They report that the majority of natural language queries from these logs required query reformulations, which indicates the need for appropriate query reformulation techniques. Satter and Sakib [150] also analyze the logs from Koders [12], construct a term-term co-occurrence matrix from the past queries, and then suggest frequently co-occurred terms from the matrix to expand a given query. Raghothaman et al. [127] analyze the click-through data from Bing search engine that contains millions of (query, URL) pairs. They mine the logical mapping between each query and API classes from corresponding URL, construct a conditional probabilistic model, and then suggest relevant API classes to reformulate a given query. Cao et al. [25] analyze the search logs from Stack Overflow Q&A website, train a sequence-to-sequence model using original and reformulated queries, and then generate a reformulated query for a given query. Dietrich et al. [40] also capture the query pairs from developers containing original and reformulated queries, mine query transformation rules, and use them to reformulate a query for traceability link recovery.

Several primary studies mine relevant API classes from API documentation [35, 102], programming Q&A websites [38, 137, 139, 156, 165], codebase, and version control history [53, 67, 69, 93, 186] to reformulate a given query. Lv et al. [102] identify relevant API classes to a query considering textual similarity between the query and candidate API documentation from MSDN, and use them to expand the query. Chatterjee et al. [35] enrich the API invocations in source code with corresponding API documentation to tackle the vocabulary mismatch problem [46] in code search.
Rahman et al. [139] first capture the co-occurrences between query keywords and API classes in the Q&A threads of Stack Overflow, and suggest relevant API classes to expand a given query. Several later studies [38, 129] extend this work. Other primary studies reformulate a query with relevant keywords [118, 134, 156] and tags from Stack Overflow [91, 92]. Nie et al. [118] capture software-specific keywords from the relevant Q&A threads using pseudo-relevance feedback, question popularity, and term weighting method, and then expand a given query for code search. Li et al. [91] construct a software-specific tag database and reformulate a given query with synonymous and relevant tags from Stack Overflow questions.

Zhang et al. [186] mine relevant API classes from ≈25,000 open-source projects to expand a given query. Lin et al. [93] use Recodoc [39], a TF-IDF based traceability recovery technique, to reformulate a given query into relevant API entities. Huang et al. [67] mine thousands of code commits from the version control history and suggest API elements from the changed code using Deep belief network (DBN) to reformulate a query. Similarly, Huang et al. [69] attempt to capture the intent of a query from the changes made to its initially retrieved code, and then expand the query with API elements extracted from these changes. While expanding a given query with relevant API classes, keywords, and tags was found useful, determining the true intent of a query still remains a challenge and unlocking that could lead to more accurate query reformulation.

About 44% (31/70) of our primary studies mine relevant API classes, keywords, and tags from various sources such as search logs, API documentation, programming Q&A site, open-source projects, and version control history. Many of these approaches are inspired by the parallel studies from Information Retrieval. While mining relevant items from these sources was found useful, determining the true intent of a search query still remains a challenge. Besides, the logs from widely used commercial search engines (e.g., Google, Bing) might not be available for public use.

(M7) Genetic algorithms: Several primary studies [78, 86, 115, 121, 125] use Genetic algorithms (GA) to reformulate their search queries. Mills et al. [115] first use a Genetic algorithm to select the near-optimal query from a bug report to support bug localization. They demonstrate that bug reports often contain near-optimal queries although there might not be any explicit hints for bug localization (e.g., program elements, stack traces), which is interesting. However, their approach uses query effectiveness to determine the fitness of query candidates. That is, the approach will need ground truth to find the near-optimal query, which makes it unsuitable for practical use. A few later studies [86, 130] revisit their findings with different datasets and reach similar conclusions. According to Rahman et al. [130], although the majority of bug reports contain near-optimal keywords, the GA-based approach might not be able to identify them from a few bug reports (e.g., ≈12%) despite the availability of ground truth information. Kim and Lee [78] perform multi-objective optimization using 15 fitness functions, and reformulate a query using Genetic algorithm to support bug localization. On the other hand, Pérez et al. [125] use Genetic algorithms to reformulate corpus documents (e.g., model fragments) in the context of feature location task. They employ four query reformulation types—expansion [141], replacement [50], reduction [56], and selection [136]—as the mutation operations and textual similarity between query and corpus document as the fitness function in their GA-based feature location. While they report benefit in reformulating the corpus documents (e.g., model fragments), the reformulation of queries was not found effective by one of their earlier work [121].

About 7% (5/70) of our primary studies use Genetic algorithms (GA) to reformulate their search queries and corpus documents. While GA-based approaches have high potential for effective query reformulations, designing an appropriate, practical fitness function still remains a challenge. Besides,
the GA-based approaches could also be costly due to their large search space constructed by all the keywords from either a bug report or a change request.

(M8) Miscellaneous: Many primary studies [15, 35, 50, 62, 75, 79, 88, 102, 107, 139, 153, 156, 183, 185] use lightweight heuristics and ad hoc metrics to reformulate their queries during code search. Zamani et al. [185] select only nouns as query keywords from a feature request to improve the feature location task. Yu et al. [183] make use of nouns, verbs, and adjectives from the texts. Shepherd et al. [153] model search queries for concern location as Verb-Direct Object (V-DO) pairs. Hill et al. [62] expand this technique with other parts of speech. Kevic and Fritz [75] consider four lightweight heuristics such as part of speech, term weight, notation and position of each keyword within a change request to separate the query keywords from the rest using machine learning.

Rahman et al. [139] capture co-occurrences between query keywords and API classes in the Q&A threads of Stack Overflow, apply three advanced heuristics to detect the relevant API classes, and then use them to expand a query for Internet-scale code search. Unlike Rahman et al., several studies [35, 93, 102] rely on textual similarity between API documentations and a query to find the relevant API classes. Other studies simply extract the structural entities such as API classes or API methods from the queries [183], test cases [88], accepted answers of Stack Overflow [156], and the relevance feedback documents [107].

Lapeña et al. [84] replace a query with the most relevant requirement specification using textual similarity to improve feature location. Wang et al. [166] extract multiple facets (e.g., structural, intent, dependency) from the initial results of a query to help developers further refine the query interactively. Chaparro and Marcus [34] adopt stepwise query reduction and suggest that removal of even only one noisy keyword can significantly improve the query. Chaparro et al. [31] perform natural language discourse analysis, identify observed behaviour (OB), expected behaviour (EB), and steps to reproduce (S2R) from a bug report (a.k.a., query), and use the observed behaviour as a reduced version of the query for localizing bugs. Zhang et al. [188] identify the neighbours of a given method in a codebase considering semantic similarity, temporal proximity, and call dependency, and expand the method’s body with the neighbouring methods to improve feature location. Sisman and Kak [159] also leverage spatial code proximity to reformulate their query for bug localization. Lu et al. [98] expand a search query leveraging several object-oriented relations (e.g., inheritance, implementation) from the source code. Kim and Lee [78] also apply several heuristics as fitness functions in their GA-based query reformulations to locate buggy code.

About 37% (26/70) of our primary studies use lightweight heuristics and ad-hoc metrics to reformulate their queries during code search. These heuristics and metrics often offer low cost alternatives to the established but costly approaches (e.g., thesaurus-based reformulation). However, they might always not be effective and thus need to be complemented with more sophisticated query reformulation techniques (e.g., term weighting).
Fig. 5. Primary studies adopting multiple methodologies in the query reformulation

Fig. 6. Overlap of the methodologies used by the primary studies

Fig. 7. Primary studies performing different types of query reformulations

and thesaurus (M3), and data mining and API recommendation (M6) – are shared by 60% (42/70) of the primary studies. Such a finding can be explained as follows. First, many term weighting methods
consider such words as *important* that frequently co-occur with query keywords in the relevant contexts such as search logs [150], change request [136, 138], source code [62, 135, 153, 157, 159], stack traces [138], and programming Q&A websites [91, 134]. Second, they might also capture such API classes to reformulate a query that frequently occur in the relevant contexts [107] and are semantically similar to the query [137, 186]. Third, the thesaurus-based approaches are connected to mining since the construction of any software-specific thesaurus involves an extensive mining of software repositories [65, 137, 139, 179, 180].

Fig. 7 shows how different types of query reformulations are used by our primary studies. We see that 56% of studies expand their queries by adding appropriate keywords whereas 13% reduce their queries by discarding the noisy keywords. On the other hand, 31% of our primary studies adopt a combination of multiple techniques to reformulate their search queries.

Fig. 8 shows how different mechanisms to collect relevance feedback are used by the primary studies. We see that 19% of studies capture explicit developer feedback on the queries for their reformulations. Unfortunately, capturing feedback from the developers could be costly and sometimes infeasible. Thus, about 26% of studies reformulate their queries using pseudo-relevance feedback without directly involving the developers. On the other hand, the remaining 56% of studies do not make use of any relevance feedback mechanism to reformulate their queries.

Fig. 9 shows how the primary studies reformulate their queries to improve code search in various problem contexts. For example, 25% of studies reformulate their queries to search various software concepts implemented in the source code. Software features are a type of concepts that can be experienced and exercised by software users. About 14% of studies reformulate their queries to locate features in the source code. Software bugs are the erroneous features that need to be removed from a software system. We see that 15% of our primary studies reformulate their queries to find bugs in the source code. On the other hand, 42% of the primary studies reformulate their queries to find relevant, reusable code segments from the Internet-scale code repositories such as GitHub, SourceForge, and programming Q&A sites such as Stack Overflow.
We also further investigate the above dimensions and report interesting patterns. Table 4 shows popular methodologies for query reformulations in various code search contexts. We see that the local code searches such as concept location and feature location are more likely to use term weighting algorithms to expand their search queries. On the other hand, Internet-scale code search techniques are more likely to use various data mining approaches, semantic relations, and popular thesauri (e.g., WordNet) to expand their queries.

**Summary for RQ1:** The primary studies on search query reformulations use eight major methodologies including term weighting and relevance feedback (40%), semantic relations, term co-occurrences, and thesaurus (46%), machine learning and query difficulty analysis (17%), and data mining and API recommendation (44%). The majority of these studies (56%) expand their queries and almost half of them capture relevance feedback on their queries. Furthermore, 57% of studies help find relevant software entities (e.g., software bugs, features) whereas the remaining studies help find relevant, reusable code examples form the Internet-scale code repositories (e.g., GitHub).

### 4.2 Answering RQ2: Evaluation & validation

Our primary studies use various methods to evaluate and validate their performance and to place their techniques in the literature. In particular, they use several evaluation methods, performance metrics, subject systems, query collections, ground truth, and validation targets. We discuss each of these evaluation dimensions with relevant statistics as follows.

**(a) Evaluation methods:** Search queries can be evaluated either through execution or without execution. Based on the presence of query execution, evaluation methods can be divided into two
Fig. 11. Top-10 popular performance metrics among the primary studies

categories—(1) pre-retrieval and (2) post-retrieval [56]. Pre-retrieval methods evaluate a search query based on its linguistic and textual aspects (e.g., coherence, specificity) without executing the query [29]. Since these aspects of a query can be captured during the indexing of a corpus, the pre-retrieval evaluation methods are lightweight and cost-effective. Unfortunately, they are unreliable as they do not analyze the results retrieved by a query [58]. On the contrary, post-retrieval evaluation methods are reliable but costly. They execute a query and compare its results with the ground truth to evaluate the performance [29, 56]. All of our selected primary studies adopt post-retrieval methods for their evaluation and validation. A few studies [57, 114] use linguistic and textual metrics to separate high-quality search queries from poorly designed queries.

Construction of ground truth can be another dimension to classify the evaluation methods. Based on this dimension, query evaluation methods can be divided into two categories—(1) re-enactment based and (2) developer study. In case of re-enactment based evaluation, ground truth information is captured from historical data [56]. For example, to assess a search query from a bug report, the changed files from corresponding bug-fix commit are considered as the ground truth [31, 138, 159]. We call this evaluation as empirical evaluation throughout the rest of article. On the other hand, in the case of developer study, the participants are responsible both for constructing the ground truth [186] and for determining relevance of the retrieved results by a query [89, 102, 150, 170, 171]. From Fig. 10, we see that 54% of our primary studies evaluate their queries empirically whereas 33% of them involve human participants for query evaluation. Interestingly, 9% of the primary studies use both types of evaluation. Although the re-enactment based method uses solid ground truth, it might always not reflect a realistic evaluation scenario since the human developers are not directly involved. On the contrary, involving human developers could also introduce subjective bias in the evaluation. Thus, a combination of empirical method and developer study is an ideal choice for evaluation, but it could be extremely costly.

(b) Performance metrics: Each of our primary studies uses one or more performance metrics to evaluate their reformulated queries. Since the majority of these studies borrow algorithms or techniques from several other domains such as Information Retrieval, Machine Learning, and Recommendation Systems, the performance metrics are also taken from these domains. We discuss these metrics and their adoption by the primary studies as follows.

Fig. 11 shows the Top-10 performance metrics adopted by our primary studies on automated query reformulation. We see that Hit@K, recall, MAP, and MRR are used by more than 30% of the
studies. Hit@K determines whether there exists any ground truth within the top K results whereas recall determines the percentage of ground truth present in the top K results. On the other hand, both MAP and MRR emphasize on the position of ground truth within the retrieved results. Another performance metric that estimates a developer’s effort to find the first ground truth within the retrieved results (a.k.a., QE), is used by 27% of the primary studies. There also exist other metrics such as NDCG and Likert that are used by less than 10% of the studies.

We also further analyze how each study uses these Top-10 popular metrics and Fig. 12 summarizes our analysis. We not only see the metric adoption ratio for each of the studies but also their distribution. According to Fig. 12-(b), at least half of the studies select two popular metrics whereas 25% of them use three or more popular metrics. On the other hand, at least 75% of the studies use at least one of the 10 popular metrics discussed above. We also found that at least 5% of the primary studies adopt five popular metrics (e.g., Hit@K, MRR, MAP, Recall, Likert) to evaluate and validate their reformulated queries. The use of these appropriate and popular performance metrics is essential to mitigate the threats to construct validity of experimental findings [184].

(c) Subject systems: Each of our primary studies uses one or more subject systems to evaluate their reformulated queries. Table 5 shows the popular subject systems used in four different working contexts. In case of concept location, we see that eight systems including Adempiere, JEdit, ECF, and FileZilla are selected by at least three primary studies. However, a total of 20 subject systems were used by at least two primary studies to evaluate their reformulated queries and to locate their desired concepts in the software code. In case of feature location, only two systems (CAF and JEdit) are selected by two or more primary studies where JEdit is common. On the other hand, in the case of bug localization, we see that 20+ systems are selected by at least two or more primary studies on average. It should be noted that the primary studies from concept location,
Table 5. Popular subject systems used for experiment in various code searches

| Code search                  | Popular systems                                                                 |
|------------------------------|---------------------------------------------------------------------------------|
| Concept location             | Adempiere (4), JEdit (4), ECF (3), FileZilla (3), eclipse.jdt.core (3), Win-      |
|                              | Merge (3), aTunes (3), eclipse.pde.ui (3), Rhino (2), eclipse.jdt.ui (2), Log4j |
|                              | (2), Tomcat70 (2), iReport (2), eclipse.jdt.debug (2), Sando (2), Jajuk (2),     |
|                              | Sling (2), JavaHMO (2), Eclipse (2), JBidwatcher (2)                            |
| Feature location             | CAF (4), JEdit (2)                                                              |
| Bug localization             | AspectJ (4), Lucene (3), ECF (3), Pig (3), eclipse.jdt.ui (3), Solr (3), Derby   |
|                              | (3), eclipse.jdt.core (3), Tomcat70 (3), OpenJPA (3), ZooKeeper (3), Book-      |
|                              | Keeper (3), eclipse.jdt.debug (3), Mahout (3), eclipse.pde.ui (3), Tika (3),    |
|                              | ZXing (2), JodaTime (2), SWT (2), Chrome (2), Eclipse (2)                       |
| Internet-scale code search   | GitHub (9), Stack Overflow (8), SourceForge (4), OSChina (2), Java2s (2),      |
|                              | Javadb (2), KodeJava (2), OpenHub (2), Google (2), JEdit (2)                   |

Fig. 13. Top-15 popular subject systems among the primary studies

feature location, and bug localization share several subject systems (e.g., ECF, JEdit, Eclipse). This might be partially explained by their methodological overlap, as demonstrated in Table 4. On the contrary, the primary studies from Internet-scale code search mostly use large-scale open-source software platforms (e.g., GitHub, SourceForge, OpenHub, OSChina) and popular programming Q&A websites (e.g., Stack Overflow, KodeJava, Java2s) for their evaluation and validation. That is, they use hundreds if not thousands of subject systems from these platforms in their experiment.

Fig. 13 further demonstrates the Top-15 popular subject systems (or software platforms) used by our primary studies across different working contexts. We see that GitHub and Stack Overflow are the most popular choices due to their large collection of open-source projects and millions of question-answering threads containing code examples. JEdit, a Java-based IDE, has also been a popular subject system among the primary studies from multiple code search contexts. Besides, several other systems such as eclipse.jdt.core, ECF, and AspectJ have been used by ≥10% of the primary studies. The use of diverse subject systems is essential to mitigate the threats to external validity of the experimental findings [184].

ACM Comput. Surv., Vol. XX, No. X, Article XX. Publication date: August 2021.
(d) **Search queries and human participants:** Each of our primary studies selects a set of search queries to evaluate their query reformulation techniques. Ideally, all queries (to be reformulated) should be collected from human developers. However, involving humans is costly and thus, only a few studies [40, 47, 56, 122] collect actual queries from the developers. The majority of primary studies (1) use historical artifacts (e.g., past bug reports) [56, 136, 159], (2) reuse queries from earlier datasets [150, 186], or simply (3) construct their queries from secondary sources (e.g., programming Q&A forums, tutorial sites) [137, 139]. These sources often deliver a good enough representative of actual search queries from human developers. Furthermore, they can provide a relatively large dataset at low cost, which is essential to conduct a large-scale empirical evaluation.

The number of used queries is an important aspect of evaluation. We collect the number of queries used by each of our primary studies and summarize our analysis in Fig. 14 and Table 6. From Fig. 14, we see that 23% (16/70) of the studies use 25 or less queries whereas 14% (10/70) of them select 26 to 50 queries for their evaluation. On the other hand, 16 studies select 150 to 500 search queries whereas 14 other studies evaluate their reformulation techniques using 500+ search queries. We further breakdown our analysis across different working contexts. From Table 6, we see that the primary studies from concept location and bug localization use the maximum number of queries, which might be partially explained by the availability of thousands of bug reports or change requests in the open-source projects. On the other hand, the number of used queries in the Internet-scale code search is generally small. Unlike that of above working contexts, these

---

**Table 6. Search queries used for experiment in various code searches**

| Code search                  | Query count | #Study | Frequent reformulation |
|------------------------------|-------------|--------|------------------------|
| Concept location             | 1–25        | 5      | Query expansion        |
|                              | 1001–5000   | 5      |                        |
|                              | 25–50       | 2      |                        |
| Feature location             | 1–25        | 3      | Expansion + reduction  |
|                              | 25–50       | 2      |                        |
|                              | 150–250     | 2      |                        |
| Bug localization             | 1000–5000   | 4      | Query expansion        |
|                              | 250–500     | 2      |                        |
|                              | 500–1000    | 2      |                        |
| Internet-scale code search   | 1–25        | 8      | Query expansion        |
|                              | 25–50       | 5      |                        |
|                              | 50–75       | 5      |                        |

**Fig. 14. Number of search queries used by primary studies**
queries cannot be easily extracted from the historical artifacts and are often collected from human developers, which might partially explain their small numbers.

Human participation is an important aspect of search query evaluation. We found that 41% (29/70) of our primary studies involve human participants in their evaluation. Fig. 15 shows our statistical analysis. We see that the majority of these studies (80%) involve 25 or less participants whereas 16% involves up to 50 developers. On the other hand, only one primary study involves 250+ human participants in a large-scale developer survey. Furthermore, we found that 52% (15/29) of the human participation is associated to the primary studies from Internet-scale code search. However, the number of developers involved in each study is small on average (e.g., 1–25). Involving human participants is always a great idea to complement an empirical evaluation. However, this involvement is often costly, which possibly explains the small number of participants recruited by the majority of our primary studies.

(e) Validation targets: Comparison with closely related existing studies (a.k.a., validation targets) is an essential step to place any work in the literature. About 99% of our primary studies validate their experimental findings by comparing with one or more existing alternatives. We collect the validation targets from each study and summarize our analysis in Table 7 and Fig. 16.

From Table 7, we see that Baseline and Rocchio’s method [141] have been frequently selected for comparison by the primary studies across three different search contexts – concept location, feature location, and bug localization. Baseline search query refers to the initial version of a query that gets reformulated by the primary studies. On the other hand, Rocchio’s method is a popular technique for query expansion borrowed from the Information Retrieval domain [29, 141]. About 33% of our primary studies compare their reformulated queries with that from Baseline and Rocchio’s methods. Besides them, other techniques such as RSV, Dice expansion, and query reduction have also been frequently selected for comparison by the primary studies across concept location and feature location. However, the primary studies that reformulate queries to localize software bugs compare with several bug localization techniques such as BugLocator [190], Amalgam+ [169], BLIZZARD [138], BLUiR [146], and Sisman and Kak [159]. On the other hand, the primary studies from the Internet-scale code search frequently compare their approaches with several ad hoc alternatives (e.g, Google search, GitHub search, Stack Overflow search) and a few established search algorithms such as BM25 [154] and TF-IDF [72]. Furthermore, several primary studies such as QECK (S34), CodeHow (S15), and RACK (S39) were also chosen by the later studies to validate their findings.

Fig. 16 further demonstrates the top 15 validation targets selected by the primary studies across different search contexts. We see that reformulated queries are often compared with their initial, non-reformulated versions (a.k.a., Baseline) to make sure that the reformulation is useful. About 18% (13/70) of our primary studies perform such a comparison. Several ad hoc alternatives such as Google search, GitHub search, and Stack Overflow search have also been popular. Furthermore, six
Table 7. Popular validation targets used for comparison in various code searches

| Code search                  | Popular validation targets                                                                                                                                 |
|------------------------------|------------------------------------------------------------------------------------------------------------------------------------------------------------|
| Concept location             | Rocchio’s method [141] (4), Baseline (4), Pessimistic constant [57] (2), RSV [140] (2), Optimistic constant [57] (2), Random classification [57] (2), Shepherd et al. [153] (2), Query reduction [56] (1), Eclipse IDE-based search (1), Dice expansion [56] (1), Conquer [142] (1), Sisman and Kak [159] (1), Manual search (1), Text-based search engine (1), GES [153] (1), FLAT3 [151] (1), Logistic regression [108] (1), Refoqus [56] (1), Kevic and Fritz [75] (1), GREP (1), LEex (1), and Hill et al. [62] |
| Feature location             | Baseline (6), Rocchio’s method [141] (2), Dice expansion [56] (2), RSV [140] (2), Query reduction [56] (2), LDA [18] (2), Eclipse search (1), LSI [148] (1), SWUM [63] (1), Shepherd et al. [153] (1), Manual search (1), VSM [72] (1), and Sridhara et al. [160] (1) |
| Bug localization             | Baseline (6), BugLocator [190] (4), BLIZZARD [138] (2), Rocchio’s method [141] (1), BRTracer [175] (1), BLUiR [146] (1), Lucene [3] (1), Lobster [117] (1), Sisman and Kak [159] (1), rVSM [190] (1), Manual search (1), BLIA [182] (1), AmaLgam+ [169] (2), Relevance model [85] (1), MULAB [189] (1), Observed behaviour [31] (1), Wang and Lo [168] (1), Genetic algorithm (1), and STRICT [136] (1) |
| Internet-scale code search   | Google search (6), GitHub search (5), BM25 [154] (5), QECk [118] (4), TF-IDF [72] (4), Stack Overflow search (3), OpenHub search (3), CodeHow [102] (3), Portfolio [109] (3), Baidu search (2), Baseline (2), OSHina search (2), Manual search (2), RACK [139] (2), VF (VSM + Item set mining) (2), SourceForge search (2), Lu et al. [99] (2), Rocchio’s method [141] (1), Ohloh search (1), NQE [97] (1), Dice expansion [56] (1), SWordnet [180] (1), Wordnet [113] (1), FIM [25] (1), HRED-qs [25] (1), GooglePS [25] (1), word2vec [112] (1), K-NN Linear (1), Thung et al. [164] (1), Codota search [5] (1), Bing search (1), Cocabu [156] (1), BIKER [66] (1), Zhang et al. [186] (1), RSV [140] (1), NLP2API [137] (1), KBCS [89] (1), Jungoloid [104] (1), NCS [97] (1), and LDA [18] (1) |

existing approaches including RSV, Dice, TF-IDF have also been used for comparison by 5%+ of the primary studies on average. Comparison with appropriate existing alternatives is essential to claim the technical superiority of a proposed technique. Unfortunately, due to the unavailability of replication packages, such a comparison might always not be possible. For example, only ≈30% of the primary studies were selected for comparison by the later studies, which might be partially explained by our findings on the replication packages in Table 15.

(f) Statistical tests: The use of statistical tests is an essential part of testing any hypothesis and drawing any conclusions. About half of the primary studies (51%, 36/70) conduct one or more statistical tests to draw conclusions about their superiority over the existing alternatives. Table 8 provides an overview of the conducted tests and Fig. 17 shows their usage statistics.

From Fig. 17, we see that Wilcoxon Signed Rank is the most popular statistical test, which has been used by 20%-+ (15/70) of our primary studies. It is a non-parametric test for two related, dependent samples, which makes it suitable for comparison between a technique and another competing alternative. Student’s t is another popular paired test that assumes normality for its samples. It has been used by 11% of the primary studies. On the other hand, Cliff’s δ is a non-parametric test to determine the effect size between any two samples, which is often accompanied by another significance test– Mann-Whitney Wilcoxon. Both these tests are suitable for the samples with unknown distribution and are used by 8%-10% of our studies. besides these four, there have
been 11 other statistical tests and each of ANOVA, Shapiro-Wilk, Vargha and Delaney’s A, and Chi-squared has been used by ≈5% of our 70 primary studies.

**Summary of RQ2:** About 54% of our primary studies use empirical evaluation, 33% perform qualitative investigation, and only 9% of the studies use both. About 80% of the studies that perform qualitative evaluation involve less than 25 developers each. Almost half of our studies select at least two popular performance metrics each for their evaluation whereas 30 different metrics have been used by all the studies. We found that 70 primary studies have selected 100+ subject systems and the studies from Internet-scale code search use less queries than the others for their evaluation. About 30% of our primary studies were also selected for comparison by the later studies. Our primary studies use a total of 15 statistical tests where Wilcoxon Signed Rank test has been adopted by more than 20% of the studies.
Table 8. Statistical tests conducted by primary studies

| Category              | Statistical test          | Overview                                                                 |
|-----------------------|---------------------------|--------------------------------------------------------------------------|
|                       | Mann-Whitney Wilcoxon     | The test determines whether any two samples are drawn from the same distribution or not. It does not assume any specific distribution for its samples. |
|                       | Wilcoxon Signed Rank      | The test determines if two related, dependent samples are significantly different or not from each other. Both samples should have equal number of elements. |
|                       | Friedman                  | The test determines if three or more samples have identical population distributions or not. The data can be either quantitative or ordinal. |
|                       | Kendall’s $\tau$          | The test determines the strength and direction of association between any two measured quantities. |
|                       | Quade                     | The test is an extension of Wilcoxon Signed Rank test for three or more related samples. |
|                       | Shapiro-Wilk              | The test determines whether a sample is drawn from a normal distribution or not. |
|                       | Cliff’s $\delta$          | The test quantifies the difference between any two samples beyond their *p*-value interpretation. |
|                       | Vargha and Delaney’s $A$  | The test quantifies the difference between any two samples using a probabilistic measure. It has a linear relationship with Cliff’s $\delta$. |
|                       | Holm’s correction         | The test is used to counteract the problem of multiple comparisons and to control the family-wise error rate. |
|                       | Paired $t$                | The test determines whether two related, dependent samples are significantly different from each other or not. It also assumes $t$ distribution for its samples. |
|                       | ANOVA                     | The acronym stands for Analysis of Variance. The test determines whether three or more independent populations are significantly different from each other, with the assumption that their distributions are normal. |
|                       | Chi-squared               | The test determines the difference between any two categorical variables in the same population. It assumes $\chi^2$ distribution for the population, and shows how the two variables might be connected to each other. |
|                       | Cohen’s $D$               | The test quantifies the difference between any two samples with an assumption of normality and homogeneity in their variance. It often accompanies $t$-test and ANOVA. |
|                       | Asymptotic General Inde-  | The test determines whether any two random variables are asymptotically independent of each other. Here, asymptotic aspect refers to a very large sample size for each variable. |
|                       | pendence                  |                                                                                   |

4.3 Answering RQ3: Challenges and limitations in automated query reformulations for source code search

Search queries often consist of a few unstructured keywords. On the other hand, software code is full of structures (e.g., syntactic relations) and has a limited vocabulary [60]. Due to this inherent difference between a query and the code, constructing the right search query is challenging.
Table 9. Challenges and limitations of the primary studies

| ID | Challenge & limitation |
|----|------------------------|
| CH1 | Noise in search query and Unsolved vocabulary mismatch problem |
| CH2 | Extra cognitive burden on the developers |
| CH3 | Lack of generalizability |
| CH4 | Issues towards practical adoption |
| CH5 | Human bias and Weak evaluation |
| CH6 | Inappropriate use of tools and Strong external dependencies |
| CH7 | Lack of sound theory |
| CH8 | Miscellaneous |

Table 10. Primary studies, their challenges and limitations

| ID | Primary studies | Total |
|----|-----------------|-------|
| CH1 | S1, S5, S7, S14, S15, S16, S19, S20, S23, S25, S26, S27, S28, S29, S35, S36, S38, S39, S40, S41, S42, S43, S44, S45, S46, S50, S51, S53, S54, S55, S57, S58, S59, S63, S64, S70 | 36 (51%) |
| CH2 | S3, S12, S13, S17, S24, S27, S28, S29, S30, S31, S33, S35, S43, S47, S48, S49, S53, S56 | 18 (26%) |
| CH3 | S1, S2, S4, S7, S8, S10, S11, S13, S14, S15, S16, S19, S22, S23, S24, S25, S27, S28, S30, S31, S32, S33, S34, S35, S36, S37, S39, S40, S41, S42, S43, S45, S46, S47, S49, S50, S51, S52, S54, S55, S56, S57, S59, S60, S64, S65, S66, S69 | 49 (70%) |
| CH4 | S2, S3, S6, S22, S27, S29, S30, S32, S33, S34, S42, S44, S48, S50, S53, S54, S61, S62, S63, S64, S66, S67, S68 | 23 (33%) |
| CH5 | S1, S4, S6, S11, S12, S13, S15, S16, S17, S18, S21, S22, S24, S26, S27, S28, S30, S31, S34, S35, S36, S37, S38, S39, S42, S43, S44, S45, S46, S47, S51, S52, S58, S60, S63, S64, S65, S66, S69 | 39 (56%) |
| CH6 | S35, S37, S48, S51, S53, S56, S61 | 7 (10%) |
| CH7 | S9, S14, S35, S44, S55, S62, S65 | 7 (10%) |
| CH8 | S8, S14, S17, S26, S27, S32, S36, S38, S45, S49, S51, S55, S57, S61, S63 | 15 (21%) |

Although each of our primary studies attempts to mitigate these challenges, they might not be entirely successful. We critically examine the methodology, evaluation, validation, and threats to validity sections from each study, and summarize their reported or implied limitations. Similar to RQ1, we use Grounded Theory approach (Section 3.5) to analyze the primary studies and detect their challenges and issues. Tables 9 and 10 show the key challenges and limitations of our primary studies. In this section, we discuss each of these key challenges in details as follows.

**CH1–Noise in the search query and unsolved vocabulary mismatch problem**: One of the major challenges of code search is the vocabulary mismatch between a query and the source code. In essence, each of our primary studies attempts to overcome this challenge. Unfortunately, almost half of them (51%) might fail due to their adoption of inappropriate methodologies and selection of noisy keywords. For example, several studies [53, 84, 93, 102, 183] rely on lexical similarity between a query and the API documentation to detect the relevant APIs and to expand the query. However, such a similarity alone might not be sufficient and false-positive API classes (a.k.a., noise) could be added to the query. Many studies [48, 89, 90, 99, 153] adopt English language thesauri such as WordNet [113] to replace or expand a query with synonyms or semantically similar words. However, WordNet is based on natural language texts rather than source code. As a result, the keywords suggested by WordNet might not be appropriate for detecting relevant source code. An existing evidence [160] also suggests that the same word could have two different meanings when used in natural language texts and in source code respectively. In other words, the keywords from
WordNet can introduce noise to a search query. Thus, the above studies on query reformulation might not be able to overcome the vocabulary mismatch problem during their code search.

Several primary studies [43, 69, 107, 118, 134, 137, 139, 156, 186] expand a search query using relevance feedback mechanisms (details in Section 2) to mitigate the vocabulary mismatch problem. However, such an expansion might also hurt the query by introducing noisy keywords. Nie et al. [118] expand a query by collecting important keywords from relevant Q&A threads of Stack Overflow using pseudo-relevance feedback. Sirres et al. [156] extract important program elements (e.g., method invocations) from such threads. Similarly, Rahman and Roy [137] extract important API classes from the Q&A threads of Stack Overflow to expand a query. All these studies apply various term weighting methods (e.g., TF-IDF [72]) to selecting their keywords. Unfortunately, the term weighting method alone might not be able to discard all noisy keywords. Several primary studies [77, 138, 188] use a bug report or a feature request as a query and collect candidate terms from source code to reformulate the query. However, they might also suffer from noisy keywords as the poor-quality bug reports or feature requests are unable to capture appropriate candidate terms from the code. Martie et al. [107] capture frequent identifier names from the source code retrieved by a given query and use them to expand the query. Huang et al. [69] mine frequently changed identifier names from the code change history to determine the intent of a search query. Eddy et al. [43] also expand a search query by repeating its structured terms (e.g., identifier names) multiple times during feature location. All these studies above greatly rely on the identifier names to expand their queries. However, the identifier names could be poor, generic, and less descriptive, which can hurt the original search queries.

A few primary studies [137, 186] learn word embeddings [181] from large corpora (e.g., Stack Overflow, GitHub). They attempt to mitigate the vocabulary mismatch problem by determining semantic relevance between a query and the source code. They strongly assume that both query keywords and code terms are present in the corpus, which might always not hold true. Rahman et al. [139] mine frequent keyword-API co-occurrences from Stack Overflow Q&A threads and use the frequent API classes to expand a query. However, such a co-occurrence might also provide noisy keywords (e.g., false-positive API classes) that can hurt the original search query.

CH2–Extra cognitive burden on the developers: Several primary studies [43, 47, 48, 62, 107, 153] impose extra cognitive burden on the developers during query reformulation. Shepherd et al. [153] first introduce an interactive approach namely Find-Concept to support query reformulation in the context of concern location. Although their approach suggests relevant keywords from multiple sources (e.g., Verb-DO pairs, synonyms), a developer is ultimately responsible to choose the right keywords and to construct the final query. Hill et al. [62] extend this work with the phrasal concept analysis. Ge et al. [48] further extend Find-Concept by recommending keywords before and after the code search. All these studies above argue in favour of developer involvement rather than a complete automation in query construction. Thus, in essence, they warrant a certain level of expertise or experience from the developers and might not be suitable for novice developers. Gay et al. [47] capture explicit feedback on a query from a developer and expand the query with the feedback to support concept location. Martie et al. [107] implement a similar idea to improve queries for Internet-scale code search. Although the developer’s feedback could be useful, capturing it regularly could be infeasible and costly. Several primary studies also warrant significant learning and cognitive efforts from the developers such as generation of test cases [88, 89], input code examples [15, 119, 171] and controlled search queries [79, 178]. To use the approach of Lemos et al. [88], a developer needs to express the information need as a complex test case. Balachandran [15] expect an input code example that can be transformed into an AST to retrieve the similar code examples. Wursch et al. [178] also expect a semi-structured query from the developers rather than a free-form query during code search. Wang et al. [166] provide multi-faceted information within
IDE to support an interactive query reformulation during feature location, which has the potential to overwhelm the developers with information overload.

**CH3—Lack of generalizability:** About 70% (49/70) of our primary studies suffer from a lack of generalizability due to their limited datasets, queries, and evaluation. The majority of these studies [47, 75, 136, 138, 166, 185] evaluate their approaches using only Java-based subject systems. According to an existing evidence [145], findings from Java-based systems might always not generalize for other systems such as C/C++ based systems. Almost all primary studies also rely on open-source systems to conduct their experiments. Only a few studies [40, 121, 125] make use of proprietary dataset for their experiments. Pérez et al. [125] collect railway software systems from their industry partner (CAF) to evaluate their work. Other studies analyze proprietary search logs from Bing search engine [127] and Stack Overflow [25] to generate their reformulated queries. There exists a little evidence suggesting that the findings from open-source systems will generalize for the proprietary software systems. Other studies [47, 56, 57, 118, 136, 138] use a single code search engine (e.g., Lucene) to evaluate their reformulated queries. However, the same query might not deliver the same performance across multiple search engines [116]. A number of studies [15, 47, 69, 75, 89, 102, 150, 153, 170, 180] also make use of a limited set of queries (e.g., ≤50) for their evaluation. A few studies [71, 75] use only one subject system. Thus, their findings might suffer from the lack of generalizability as well. Several primary studies [53, 178, 183, 185, 186] might not perform equally for all types of given queries. That is, they might return good reformulated queries only if the given queries are carefully designed in the first place, which, in essence, defeats the overall purpose of automated query reformulations.

**CH4—Issues towards practical adoption:** About 33% (23/70) of our primary studies suffer from major issues that might prevent their adoption by software developers. Several studies [84, 106, 124, 150] make use of Latent Semantic Indexing (LSI) and term-document co-occurrence matrix to reformulate their search queries. Although the LSI might be useful to mitigate the vocabulary mismatch problem [46], it suffers from the curse of dimensionality problem. That is, the semantic space needs to be stored in a large, sparse matrix, which can be both costly and inefficient. Besides, the semantic space needs to be updated frequently for each incoming query before its execution, which is time-consuming and undesirable. Several approaches require non-trivial items such as test cases [88], code examples [15, 171], and salient keywords [48, 153] from developers, which could make the developers reluctant to use these approaches. Several studies [34, 86, 115] make use of ground truth information to identify appropriate search keywords from a bug report. Although these studies might be useful to provide a proof of concept, they are not suitable for a practical use since the ground truth is not generally known during query construction [115]. Several other studies involve costly computations such as genetic algorithms [78, 125], frequent AST construction [67], or ontology construction [71], which might outweigh the benefit of query reformulation. A few studies [16, 43, 185] need significant configuration management or parameter tuning, whereas the others use proprietary, publicly unavailable datasets (e.g., Bing search logs [127], Stack Overflow search logs [25]). There also exist other approaches [34, 47, 107] that involve the developers in costly trials and errors. Furthermore, 47% of our studies do not have a replication package and thus might not be replicated properly. All these constraints and issues might prevent the developers from using ≈33% of our primary studies to reformulate their search queries.

**CH5—Human bias and weak evaluation:** About 56% (39/70) of our primary studies might suffer from subjective bias and weak evaluation. Several studies [15, 88, 89, 107, 150, 186] evaluate their approaches involving undergraduate or graduate students in a laboratory setting. These participants neither represent the real users (e.g., professional developers) nor are adequate in

\[^{4}http://mlwiki.org/index.php/Vector_Space_Models\]
number. For example, 80% of our studies that engage humans have less than 25 participants each (Fig. 15). These participants were also responsible for both the ground truth preparation and the query evaluation. Thus, there exists a good chance of subjective bias in the evaluation since the appropriate measures such as inter-rater agreement or pooling method [186] were not always taken. Replication of an existing work might also introduce implementation bias especially if the replication package is not available. A few studies [56, 62, 137–139, 170] might suffer from this bias. Many primary studies [43, 75, 135, 159, 170, 185] report their best performances using their well-tuned configurations against certain subject systems (Table 5). Unfortunately, the same performance might not be guaranteed for other subject systems with the same configurations. Several studies [89, 124, 125] do not make use of popular metrics whereas the others [25, 71, 75, 97, 177] do not compare with established baselines from the literature. Besides, 49% of the primary studies do not perform any statistical test, which is an important step of rigorous validation.

**CH6–Inappropriate use of tools and external dependencies:** Over the last two decades, many tools, methods and technologies from Information Retrieval (IR) and Natural Language Processing (NLP) domains have been used to solve Software Engineering problems. Although they work with their default configurations, they need to be configured properly for Software Engineering problems for an optimal performance. They also need to be used with caution [56, 116]. For example, several studies make use of WordNet [53, 99, 153, 185] and Stanford CoreNLP POS tagger [139, 183] in the context of Software Engineering problems. Both these tools were trained on regular English texts (e.g., Wall Street Journal) rather than the source code. Thus, the semantic inferences and POS tag predictions made by these tools could be either noisy or even inaccurate, as confirmed by the existing evidence [103, 160]. Many primary studies [47, 56, 74, 93, 118, 121, 170, 183] also use TF-IDF [72] as a de facto term weighting method to select important keywords from source code. However, TF-IDF was originally designed for regular texts (e.g., news articles, legal documents) rather than source code. Regular texts and source code differ in their syntax, structures, semantics, and vocabulary [60]. Thus, the adoption of TF-IDF to select keywords from source code might not be appropriate. Several studies heavily depend on third-party items such as code change history [69, 77], semantic web technologies (e.g., SPARQL) [71, 178], tags from Stack Overflow [118], and input code examples [15, 171], and thus might be negatively affected by their absence.

**CH7–Lack of sound theory and strong intuition:** Several primary studies [31, 69, 90, 121] make counter-intuitive or unsubstantiated theoretical assumptions in their approach. Huang et al. [69] attempt to determine a query’s intent using the code change history. First, they mine the code change history of 631 GitHub projects and construct a database that contains thousands of <code segment, changes> pairs. Then they analyze the code segment retrieved by a query, identify its frequently changed identifier names using the database as query intent, and then reformulate the query with the intent. However, the idea of capturing a query’s intent with the changed code lacks a strong theoretical justification. A few other studies [67, 177] might also have similar issues. According to Chaparro et al. [31], an ideal bug report contains three parts- (a) observed behaviour (OB), (b) expected behaviour (EB) and (c) steps to reproduce (S2R) the bug. Although Chaparro et al. demonstrate that OB can be used as a reformulated query for bug localization, the idea of using OB over EB or S2R lacks a sound theoretical justification. Mills et al. [115] also could not reproduce the findings associated with OB. According to Lemos et al. [90], WordNet could be effective for query expansion during interface driven code search (IDCS) rather than keyword based code search (KBCS). However, such a claim was also not theoretically justified.

**CH8–Miscellaneous:** About 21% (15/70) of our primary studies suffer from miscellaneous issues and limitations. Zhang et al. [188] do not respect the chronology of bug reports through their use of 10-fold cross validation. That is, they might have identified previous bugs from the history by training their approach on the later bug reports, which is problematic. The primary studies on
query quality prediction [57, 114] make use of lexical and syntactic properties of a query but mostly overlook the semantic aspect. They also target binary classification (e.g., good query and bad query), which might not be sufficient since the search queries can have non-binary quality levels (e.g., very good, good, bad, very bad). Vinayakarao et al. [165] extend the vocabulary of source code by annotating program entities (e.g., `int arr[]`) with their equivalent concepts from natural language texts (e.g., integer array). However, their approach might fall short for the nested program entities with subsumption problems. Huang et al. [67] consider two code segments similar if they have similar AST nodes. However, their approach does not guarantee that both code segments have similar semantic, i.e., implement the same programming task. Huang et al. [69] also assume that a query’s intent can be better expressed with the changed identifier names rather than focusing on the semantics of the query, which is counter-intuitive.

**Summary for RQ3:** Almost half of our primary studies might add unexpected noise to a query during its reformulation whereas 26% of them impose extra cognitive burden on the developers. The majority of these studies also suffer from a lack of generalizability due to their limited datasets, queries, and evaluation. About 33% of our studies suffer from the issues that might prevent their adoption by software developers whereas more than half of them could have subjective bias in their evaluation. Furthermore, several studies heavily depend of third-party items for their query reformulations and the others make counter-intuitive assumptions in their approaches.

### 4.4 Answering RQ4: Evidence of existing researches on automated query reformulation

We collect publication year, venue, and authors of each primary study to construct the evidence of existing researches on automated query reformulations. Such an evidence not only demonstrates the interest and enthusiasm around the research topic but also shows the quality of the work being produced and published. Figures 18, 19, Tables 11, and 12 present various statistics on the conducted studies over the last 15+ years.

Fig. 18 shows the frequency of publications on automated query reformulations per calendar year. We see that the first work on automated query reformulation got published back in 2004. Marcus et al. [106] first introduced query reformulation in the context of concept location where they used...
Table 11. Existing publications on query reformulation targeting source code search

| Year | Published studies | Total |
|------|-------------------|-------|
| 2004 | S50 (WCRE)        | 1     |
| 2006 | S18 (LNCS)        | 1     |
| 2007 | S43 (ASOD)        | 1     |
| 2009 | S12 (ICSE), S30 (ICSM), S55 (LNCS) | 3 |
| 2010 | S48 (ICSE), S51 (ASE) | 2 |
| 2011 | S3 (IST), S31 (ICPC), S47 (ASE) | 3 |
| 2012 | S8 (ASE)          | 1     |
| 2013 | S6 (MSR), S10 (ICSE), S24 (ICSE), S28 (MSR), S52 (ASE) | 5 |
| 2014 | S1 (IST), S4 (ASE), S11 (ICSE), S27 (EMSE), S42 (MSR), S45 (MSR) | 6 |
| 2015 | S13 (SCAM), S14 (ASE), S15 (ASE), S33 (ICSME), S36 (SANER), S46 (ICSE), S53 (SANER) | 7 |
| 2016 | S5 (Internetware), S17 (VLC), S23 (SPLC), S29 (ICSE), S34 (TSC), S37 (Internetware), S38 (ASE), S39 (SANER), S41 (ICSE), S56 (ASE) | 10 |
| 2017 | S2 (ICCIDT), S20 (TSE), S21 (JSEP), S22 (ASE), S26 (ASE), S32 (TOSEM), S40 (SANER), S44 (ICSME), S49 (WSDM), S54 (JSEP) | 10 |
| 2018 | S7 (EMSE), S9 (DKE), S16 (FCS), S19 (ICSME), S25 (FSE), S35 (SPE), S60 (Access), S67 (ICSME), S68 (ICSME) | 9 |
| 2019 | S57 (IST), S58 (MAPL), S59 (AUSE), S61 (SAC), S63 (SPE), S65 (Access), S69 (TSE), S70 (ICPC) | 8 |
| 2020 | S62 (IST), S64 (Access) | 2 |
| 2021 | S66 (ICSE)        | 1     |
|      |                   | **70** |

word similarity derived from *latent semantic space* to automatically reformulate a given search query (e.g., “font style”). From 2005 to 2012, there had been a few studies targeting query reformulation, and on average 2-3 papers got published every year. However, the query reformulation received much attention as a research topic during 2013 and onward. We believe that such an attention or enthusiasm could be partially attributed to a few contemporary doctoral theses on query reformulations and code searches [55, 61]. Since 2013, there has been a significant increase in the research activities and publication counts. For example, during the last six years (2015–2021), 47 studies on query reformulations were published where 2016 and 2017 were the most productive years (e.g., 10 studies per year). It also should be noted that ≈80% of our selected primary studies were conducted during the last nine years. Table 11 further breaks down the statistics and shows the venues where each study was published. Given the rise of large open-source software systems (e.g., Apache, Eclipse), extra-large code repositories (e.g., GitHub), and popular programming Q&A sites (e.g., Stack Overflow), code example search and automated query reformulations have become highly relevant and attractive research topics. The empirical findings in Fig. 18 above also support our observation about conducted researches on automated query reformulations.

Fig. 19 shows the publication count for each venue. Table 12 further breaks down the statistic for each venue across different calendar years. We see that 50% (35/70) of our selected studies got published in the top venues of Software Engineering such as ICSE, ASE, ICSME, SANER, and MSR. About 28% of them (20/70) also made into the flagship conferences such as ICSE (A*) and ASE (A*). Many of these 70 studies were extended later and got published in the reputed journals such as
EMSE and JSEP. There were also a number of Journal-First studies on query reformulations that got published in several high-quality journals such as TSE, TOSEM, EMSE, JSEP, SPE and TSC. While ASE and ICSE communities pioneered this research topic, other communities (e.g., ICSME, MSR, SANER) have also been catching up to speed especially in the last few years. We also see such an evidence of significant research activities from the Table 11.
We also identify the authors from each of our 70 primary studies and design a word cloud using their first and last names. We found a total of 172 Software Engineering researchers who contributed to the research of automated query reformulations to improve code search. In Fig. 20, the size of each of the names indicates their relative publication counts on automated query reformulations.

The demand for low-cost, accurate, and easy-to-use solutions for code search is likely to grow in the coming years due to the constant growth of open-source repositories (e.g., GitHub, BitBucket) and the increasing complexity and size of modern software systems. Reformulation of a search query is an inevitable step of code search regardless of the search context. Thus, automated query reformulation is likely to be an active, interesting research topic for the years to come. However, given our identified challenges and limitations (Section 6), we suggest that the community should focus on designing more practical solutions that can be easily integrated by software developers in their regular workflows.

**Summary of RQ4**: To date, according to our investigation, a total of 70 primary studies on automated query reformulations supporting code search have been conducted. About 80% of them were conducted in the last nine years. This trend not only indicates the timeliness or relevance...
4.5 Answering RQs: A closer look at local code search and Internet-scale code search

Local code search such as bug localization, concept location, or feature location attempts to find the location of a software bug, a concept, or a feature respectively within the code of a software system. On the other hand, Internet-scale code search attempts to find out relevant, reusable code segments from thousands of software projects stored at Internet-scale code repositories (e.g., GitHub). More specifically, while the former detects code locations that need to be modified, the latter finds code segments that can be reused. Although these two searches differ in their working context and end goal, they could overlap in several other aspects. In this systematic literature review, we analyze 70 primary studies where 58% of them reformulate queries for local code searches and the remaining 42% studies attempt to improve Internet-scale code search through query reformulations (e.g., Fig. 9). We refer to these two groups of studies as QR$_{LC}$ and QR$_{IC}$ respectively. A closer look at the query reformulations supporting these two searches can provide valuable insights. In this section, we compare and contrast between these two groups of primary studies based on their underlying methodologies, algorithms, evaluation methods, challenges, and limitations as follows.

**Underlying methodologies and algorithms in query reformulation.** Among 70 primary studies, 40 studies (QR$_{LC}$) reformulate queries to support local code searches (e.g., bug localization, concept location) whereas the remaining 30 studies (QR$_{IC}$) target Internet-scale code search. Fig. 21 shows the methodologies adopted by these two groups of primary studies. We see that term weighting and relevance feedback mechanism are frequently used to select keywords and to reformulate queries by the primary studies associated to local code searches. About 23% of them adopt this methodology whereas the ratio is only 11% for the studies from QR$_{IC}$. In local code search, there are several software artifacts such as bug reports and feature requests from which the
Fig. 23. Relevance feedback by the primary studies supporting (a) local code search, and (b) Internet-scale code search

Keywords can be chosen based on their weight or appropriateness, which explains the popularity of term weighting methods. On the other hand, in the Internet-scale code search, query keywords are often chosen by the developers on the fly according to their information need. Semantic relations, word co-occurrences, and thesauri have also been used by 15% and 25% of the primary studies from these two groups respectively. However, the most popular methodologies for query reformulations that support Internet-scale code search are data mining and API recommendation. About 28% of the studies from QRICS adopt this methodology whereas the ratio is only 11% for the counterpart. In Internet-scale code search, search queries are often short (e.g., 2–3 keywords) and thus they require an expansion to retrieve the desired code [14, 144]. Many primary studies [38, 66, 137, 139, 186] expand these queries with relevant API classes or methods mined from large software repositories (e.g., GitHub, Stack Overflow), which explains the popularity of API recommendation methodology (i.e., M6). Several studies [89, 90, 92, 99, 165] also expand these short queries with synonyms or similar words from well-known thesauri such as WordNet [113]. This explains a high usage ratio (25%) of thesaurus based methodology (i.e., M3) by the studies from QRICS (Fig. 21-(b)).

Fig. 22 shows the types of query reformulations performed by primary studies supporting both local and Internet-scale code searches. We see that query expansion is very common for both groups (e.g., 41%–77%). However, queries are more frequently expanded in the Internet-scale code search due to their short length [12]. Query reduction is less common for both groups of studies. Interestingly, in local code search, a significant number of queries (e.g., 44%) receive other types of reformulations such as the combination of query expansion, reduction, or replacement.

Fig. 23 shows how relevance feedback is used to reformulate queries by the primary studies. We see that query reformulations in the local code search use comparatively more feedback. About 54% of them use either direct feedback from developers or pseudo-relevance feedback from a search engine. On the other hand, only 33% of studies from the Internet-scale code search (QRICS) use any form of feedback to reformulate their queries. It should be noted that these QRICS studies capture less feedback although they involve human developers more frequently than their counterpart (check Section 4.2-(d)). Any local code search takes place on a single software project whereas the Internet-scale code search deals with thousands of cross-domain software projects. Thus, the results from Internet-scale code search might be noisier, and capturing relevance feedback against them could be costly and challenging. Such a high cost might have discouraged any form of relevance feedback. According to Fig. 23, 46%–67% of studies from both groups did not use any form of relevance feedback to reformulate their search queries.

**Evaluation and validation in query reformulation.** Among 70 primary studies, 36 of them use empirical evaluation, 23 involve only human participants whereas 6 studies conduct both empirical and developer evaluation. On the other hand, the remaining 6 studies conduct case studies to evaluate their approaches. Fig. 24 shows a breakdown of these statistics for two groups of primary studies. We see that 61% of the studies supporting local code search evaluate their queries with empirical methods using historical artifacts (e.g., bug reports, version control history). On the
contrary, 37% of the studies supporting Internet-scale code search perform empirical evaluation. However, their popular evaluation method is developer study. About 50% (33% + 17%) of the studies from the Internet-scale code search involve human developers in their evaluation. Such a ratio is 34% (32% +2%) for the counterpart. Existing primary studies from local code search often construct their dataset and ground-truth from the version control history (a.k.a., bug-fixing commits) of open source projects that are widely available in recent years. On the contrary, the historical data from Internet-scale code search such as search engine logs are proprietary artifacts that might not be publicly available. Thus, these studies often involve human developers to evaluate their queries.

Fig. 24 further contrasts between two groups of our primary studies based on their use of popular performance metrics. We see that Hit@K, Recall, MAP and MRR are generally popular among both groups. However, the studies from local code search are more interested about query effectiveness (QE). In local code searches such as concept location, bug localization, or feature location, the quick identification of the first correct result is important. The underlying idea is that once the first correct document is located, the other relevant documents can be easily located through their structural dependencies (e.g., call graphs) within a software project [55]. The same might not be true for the Internet-scale code search and 7% of their studies use QE. Their search results are retrieved from thousands of cross-domain software projects and static dependencies might not be applicable beyond a single project. We also see that unlike $Q_{LCS}$, $Q_{ICS}$ studies are more interested to developer-oriented metrics such as spent time, NDCG, and Likert that capture a developer’s impression on the results retrieved by a query.

**Challenges and limitations in query reformulation.** Since the studies supporting local code search ($Q_{LCS}$) and the studies supporting Internet-scale code search ($Q_{ICS}$) overlap in their methodologies (check Fig. 21), they might suffer from similar types of challenges or limitations. Fig. 26 summarizes our analysis on this aspect. We see that the studies from both groups are similarly affected by several challenges and limitations (Table 9). About 17% of $Q_{LCS}$ studies could suffer from noise in their queries whereas the corresponding statistic is 21% for the $Q_{ICS}$ studies. In local code search, developers can select keywords from existing artifacts (e.g., bug reports) whereas they need to select keywords on the fly during Internet-scale code search. Thus, the later queries need
more extension (Fig. 23) and might have a higher chance of getting noisy keywords. The existing query reformulation approaches in the local code search impose more cognitive burden on the developers than their counterpart. Changes to wrong code often have major negative consequences (e.g., subsequent bugs) [20, 21]. Thus, precisely locating the source code of interest is a major step in the local code searches. On the contrary, developers might accept the partially relevant code examples during Internet-scale code search, which they can customize based on their needs. Both groups of studies also suffer from a lack of generalizability, which is the most prevalent issue. The findings from 23%–28% of the studies from both groups might not be generalizable, as acknowledged or implied by their authors. Although existing software artifacts (e.g., software, bug reports, version control history) are extensively used to evaluate query reformulations in the local code search, the majority of them target Java-based systems. According to an existing evidence [145], the findings from Java-based systems might not generalize for other systems such as C/C++ systems. On the other hand, the datasets used to evaluate query reformulations in the Internet-scale code search are either small or proprietary in nature. For example, 60% of the QR\(_{ICS}\) studies use 75 or less number of queries each in their evaluation. A few studies make use of proprietary items such as search logs from Bing [127] and Stack Overflow [25], which might not be available for a public reuse and replication. From Fig. 26, we also see that both groups suffer similarly with the remaining issues including human bias, weak evaluation (CH5, Table 9), extreme external dependencies (CH6, Table 9), and a lack of sound theoretical justification (CH7, Table 9).

Table 13 further compares and contrasts between the query reformulation approaches supporting local code search and Internet-scale code search. We see that they are different from each other in terms of the majority of dimensions such as query length, keyword selection methodology, operational context, search intent, and even in the ranking priority. For example, in a local code search, developers attempt to precisely locate the source code entity that needs a change. Changing the wrong code entity can have serious consequences [20, 21]. On the contrary, in the Internet-scale code search, developers attempt to locate the most relevant code example that can be reused to implement a programming task. Despite these differences, the two groups of studies are also surprisingly similar across several dimensions such as their performance metrics (e.g., Hit@K, Recall, MAP) and experienced challenges or limitations (e.g., lack of generalizability).

**Summary of RQ5:** About 58% of our primary studies support local code searches (e.g., concept location, bug localization) whereas the remaining 42% focus on the Internet-scale code search. The query reformulation approaches supporting these searches share their methodologies, algorithms, performance metrics, and even suffer from similar challenges and limitations. Appropriate keyword selection is an essential step of any query reformulation. In local code search, keywords are often selected using various term weighting methods (e.g., TF-IDF) whereas data mining and API recommendations are adopted in the context of Internet-scale code search. The primary studies from these two searches also differ in their reformulation types, evaluation methods, and human
Table 13. Comparison between query reformulations for local code search and Internet-scale code search

| Dimension                | Local code search (LCS)                                                                 | Internet-scale code search (ICS)                                                                 |
|--------------------------|----------------------------------------------------------------------------------------|-------------------------------------------------------------------------------------------------|
| Query & search intent    | Preprocessed version of a bug report, a change request, or a feature request            | Developer chosen keywords                                                                       |
| Initial query length     | Comparatively long                                                                      | Short (e.g., 1-3 keywords [14])                                                                 |
| Keyword selection        | Performed with term weighting and relevance feedback (M1)                               | Performed with data mining and API recommendation (M6)                                           |
| Query reduction          | Used by 15% of the studies                                                              | Used by 10% of the studies                                                                      |
| Operational context      | Software change tasks (e.g., bug fixing)                                                | General-purpose programming tasks                                                                |
| Search intent            | Precisely locating an entity of interest                                                | Locating the most relevant code example to a programming task                                    |
| End goal                 | Making changes to the existing code                                                     | Implementing a programming task                                                                 |
| Relevance feedback       | Used by 54% of the studies                                                              | Used by only 33% of the studies                                                                  |
| Corpus, ground truth, & evaluation | Source documents (e.g., classes, methods) from a single software project (e.g., JEdit [47]) | Source documents from thousands of software projects (e.g., Sourcerer [13])                      |
| Ground truth construction| Commits from the version control history [56, 138]                                      | Relevance judgement from developers using pooling method [186]                                  |
| External resources       | Past bug reports, bug-fixing history, author history, knowledge graph [93]             | WordNet, Stack Overflow, GitHub                                                                  |
| Ranking priority         | First correct result should rank high                                                  | All correct results should rank high                                                             |
| Popular metrics          | Hit@K, Recall, MAP                                                                      | Hit@K, Recall, MAP                                                                               |

| Miscellaneous             |                                                                                       |                                                                                                  |
|---------------------------|----------------------------------------------------------------------------------------|-------------------------------------------------------------------------------------------------|
| Major challenges          | Lack of generalizability, developer bias, weak evaluation, and noisy queries           | Lack of generalizability, developer bias, weak evaluation, and noisy queries                      |
| Experience required       | Domain knowledge (e.g., design, architectures) of a system under study                  | Relevant API libraries                                                                           |
| Research evidence         | 58% of the primary studies                                                              | 42% of the primary studies                                                                       |

participation. However, both groups of studies suffer from a common set of issues such as a lack of generalizability, weak evaluation, and noisy queries.

5 FUTURE RESEARCH DIRECTIONS

To date, 70 primary studies supporting code search through query reformulations have been conducted. Among them, 40 studies focus on local code searches that involve software change tasks (e.g., bug localization, feature location, concept location). On the other hand, 30 studies reformulate queries to support the Internet-scale code search. Despite these significant number of studies, we believe that there is still room for future work and many novel dimensions are yet to be explored. Based on our empirical and qualitative analyses, we present a list of future research directions that can improve query reformulations and support code search as follows.

5.1 Promises of keyword selection algorithms in IR-based bug localization

Information Retrieval (IR) has been extensively used in at least 20 Software Engineering tasks including bug localization [56]. Bug localization is a type of local code search where bug reports serve as a major source of queries. A few recent studies [76, 83, 167] point out potential biases and limitations in this local code search. According to them, IR-based localization is only good when the bug reports contain localization hints (e.g., program entity names). However, these empirical studies
use the whole texts of a bug report (i.e., title + description) as a query and overlook the potential of optimal search queries. Mills et al. [115] recently conduct a large-scale empirical study using Genetic Algorithms and present a positive evidence for both bug reports and IR-based localization. They suggest that the bug reports contain sufficient keywords to return the faulty source code at the top-ranked positions. In particular, the IR-based localization can succeed 67%–88% of the times even if their bug reports do not contain any localization hints [115]. Similar finding has been reported by a recent work of Rahman et al. [131]. Thus, the real challenge becomes extracting the right search keywords from a bug report regardless of its localization hints.

Appropriate term weighting algorithms can play a major role in selecting the search keywords. To date, existing term weighting studies exploit each term’s frequency [56, 72, 75, 141], statistical relations or syntactic dependencies [136, 138], or occurrence probability [26, 157, 159] as a proxy to its relative importance (a.k.a., term weight). However, many of these algorithms were designed for regular texts in the Information Retrieval domain and might not be well configured for bug reports containing a mix of regular texts and structured entities (e.g., stack traces, test cases, program elements). Thus, more sophisticated and efficient term weighting algorithms are required to improve the search queries in IR-based bug localization.

5.2 Promises of Genetic Algorithms in IR-based bug localization

Mills et al. [115] demonstrate that Genetic Algorithms (GA) are capable of generating optimal or near-optimal queries from a bug report (that can localize faulty code) when the ground truth information is provided. Unfortunately, in practice, the ground truth is not known when a search query is constructed to find a bug. Thus, the ground truth should not be a part of fitness function when designing a GA-based query reformulation technique. In other words, we need a fitness function that can identify the better one among two different queries without needing their ground truth, which is a major challenge. Several studies [56, 57, 114, 116, 135] make use of query difficulty metrics (e.g., specificity, coherency [29]) to predict the best one from a list of queries. However, these metrics have non-linear relationships with the query quality, which warrants special care during their use. One possible way to address this is to formulate the query construction as a multi-objective optimization problem, as demonstrated by Kim and Lee [78]. They design a complex fitness function that attempts to optimize 15 objectives while reformulating a bug report into an optimal query for bug localization. Future work can combine query difficulty analysis and machine learning to design a fitness function that can be realistically used by the Genetic algorithms. IR-based localization has not been widely adopted by the software practitioners due to its inherent limitations (e.g., low accuracy) [167]. However, we believe that IR-based bug localization equipped with GA-based optimal queries can make a difference in this regard.

5.3 Word embedding technology in query reformulation and code search

Several primary studies [48, 62, 90, 99] use English language thesauri such as WordNet [113] to expand a query with synonyms or semantically similar words. However, Sridhara et al. [160] demonstrate that the same word can have two different meanings when used in source code and when used in regular texts. Since WordNet is based on regular English texts, its suggested keywords might not be appropriate for source code search. Several studies [65, 179, 180, 184] construct software-specific thesauri analyzing various software repositories such as open-source systems and programming Q&A site. Unfortunately, construction and update of these thesauri are costly, and their effectiveness in the query reformulation is not well tested. Recently, a few studies [36, 137, 181, 186] make use of word embedding technology to determine semantic similarity or relevance between any two software specific words. A few of them [137, 186] use embeddings to reformulate queries for code search and also report positive findings, which is inspiring.
Word embedding technology approximates the meaning of a word using a high dimensional numeric vector \[112\]. The vector places the word as a single co-ordinate within a high dimensional semantic space (a.k.a., semantic hyperspace). Such a convenient approximation of word semantics reduces various natural language processing tasks (e.g., synonym detection, semantic distance calculation) into simple algebraic or geometric operations. Thus, we believe that this technology has lots to offer to various text retrieval tasks including query reformulation, code search, and even bug understanding. For example, the semantic hyperspace constructed using word embeddings could be useful to (a) understand erroneous and expected software behaviours from a bug report \[31\], (b) design appropriate fitness functions for the GA-based query reformulations \[115\] and (c) design more effective keyword selection algorithms to overcome the vocabulary mismatch problems.

### 5.4 Improving term weighting algorithms with useful contextual information

Determining importance of a term within a body of texts (e.g., bug report, source document) has long been recognized as a major challenge \[16, 75\]. TF-IDF \[72\] is a term weighting algorithm that has been extensively used both in Information Retrieval and in Software Engineering. The algorithm determines a term’s importance without considering its contextual information (e.g., surrounding terms). However, a term’s semantics are often determined by its contexts \[113, 184\]. Besides, several primary studies demonstrate the benefits of incorporating contexts in their term weighting algorithms. To date, several contextual items such as spatial code proximity \[159\], positional relevance \[138, 157\], term co-occurrences \[48, 62, 135, 136, 153\], syntactic dependencies \[136\], time-awareness \[185\], and structural awareness \[16, 43, 138\] have been used. However, these contexts were employed by various studies in isolation. Future work can investigate how combining these contextual items might benefit the traditional term weighting algorithms such as TF-IDF \[72\]. Genetic Algorithms can be used to optimize the relative weights of these contextual items during their combination \[184\]. Machine learning can also be used to determine complex, non-linear relationships between these contexts and a term’s importance \[75\].

### 5.5 Promises of Stack Overflow in query reformulation and code search

Software developers frequently use natural language queries to find relevant code on the web \[13, 14\]. Each of their queries contains a few keywords that might not be able to capture their information need. Existing findings \[14, 89, 137, 186\] suggest that inclusion of relevant API classes or methods in this query can improve its chance of retrieving the relevant code. A few studies \[118, 139, 156\] analyze Q&A threads from Stack Overflow to expand a natural language query and to support the Internet-scale code search. For example, Rahman et al. \[139\] analyze the co-occurrence of query keywords in the title of a question and the API classes in the corresponding accepted answer, and expand a query with frequently co-occurred API classes. (Such co-occurrences can also be leveraged in machine translation \[97, 127\]). Similarly, Stack Overflow could be leveraged to transform a feature request into relevant API classes and methods, which can lead to an implementation of the requested feature \[163\]. Since the Q&A site deals with thousands of API programming issues, bugs, and corresponding code level solutions, it can be used to better explain any bug with appropriate terminologies or code examples. Besides, in Stack Overflow, the contents are curated by a large technical crowd and their quality is controlled through a voting mechanism. Thus, we believe that Stack Overflow has a lot to offer to not only code searches but also to other maintenance activities.

### 5.6 Query worsening minimization

Automated query reformulations have both benefits and costs. Existing studies suggest that automated query reformulations might improve the performance of a search by up to 20% \[100, 185\]. However, several studies \[62, 121, 155\] also question the idea of complete automation in query
reformulation. Automated reformulations might add noisy keywords that could drift a query away from its original topic [155]. Thus, we need such solutions that can maximize the benefits and minimize the costs of automated query reformulations.

Haiduc et al. and colleagues [56, 57] determine the quality of each given query using 28 query difficulty metrics [29] and reformulate the poor queries to improve concept location tasks. Later studies [131, 135] also adopt the query difficulty metrics to separate good queries from poor ones using machine learning. However, any classifier could be prone to false-positives, and any wrong classifications will lead to costly, inappropriate reformulations to a search query. To minimize the cost, Rahman and Roy [135] even return the original query using machine learning if the reformulated queries are not good enough. Unfortunately, the risk of query worsening due to automated reformulations still remains. Like earlier studies [40, 48, 62, 170], we thus believe that human cognitive power could be leveraged in this case. According to Dietrich et al. [40], human developers might perform well in removing irrelevant terms from a search query, but could perform poorly in adding the new relevant terms. Relevant keywords could be hidden within thousands of identifier names (e.g., classes, methods) of a system’s codebase. Future work can combine the strengths of both human developers and automated tools to minimize the cognitive burdens on the developers and to reduce the costs of inappropriate query reformulations.

5.7 Improving pseudo-relevance feedback (PRF)
Capturing developers’ feedback on a query by analyzing its results can be costly and sometimes impractical. Several primary studies [56, 134, 135, 159] thus use pseudo-relevance feedback (PRF) as a feasible choice to reformulate their queries. That is, they naively assume the top K documents retrieved by a given query as relevant and then extract important keywords from them to reformulate the query. These approaches have been reported to improve upon the given queries [56, 100]. However, such a feedback mechanism might not help much if the given queries are already poor [138]. Then the retrieved documents are likely to be irrelevant, which will lead to noisy reformulated queries. While PRF has been frequently used in local code searches (e.g., concept location, bug localization), not many studies except a few [69, 107] use PRF in the Internet-scale code search. In the Internet-scale code search, the results are retrieved from thousands of open source projects and thus could be noisy or hard to comprehend. Future work can focus on designing such a relevance feedback mechanism that is cheap, light-weight, adaptive to query quality and yet reliable enough to deliver the appropriate keywords for improving the given queries.

5.8 Word embeddings technology for bug understanding
Chaparro et al. [33] extract three important components – observed behaviour (OB), expected behaviour (EB), and steps to reproduce (S2R) – from a bug report and identify 154 discourse patterns using the Grounded Theory approach [51]. They also use OB as a reduced version of the original query (i.e., bug report) to localize software bugs [31] and to find duplicate bug reports with Information Retrieval approaches [32]. Identification of these components and patterns is a major step towards bug understanding and diagnosis. While these patterns and components were extracted using Grounded Theory, they could be further investigated and possibly extended using duplicate bug reports and word embedding technology. Duplicate bug reports are likely to refer to the same bug and to share their underlying semantics such as observed behaviour or expected behaviour. That is, if the terminologies used in the duplicate bug reports are visualized within a semantic hyperspace using corresponding embedding vectors (check Section 5.3), they might provide further insights about their observed behaviours, expected behaviours, and even their discourse patterns. Given the geometric representation of a semantic hyperspace, future work can investigate whether those discourse patterns could be explained with geometric theories. A solid
understanding of these patterns could encourage novel tools both for bug understanding and even for bug fixing.

5.9 Promises of PageRank Algorithm in term weighting and source code retrieval

Rahman and Roy adapt PageRank algorithm [23] from Information Retrieval domain to reformulate queries in several Software Engineering contexts such as bug localization [138], concept location [133–136] and Internet-scale code search [137]. Other studies [94, 109] make use of PageRank algorithm to rank code examples during their code search. PageRank operates on a graph-based structure, implements a notion of voting or recommendation, and then detects the most important nodes from the graph using recursive computations [17, 111]. Since source code is full of structures, entities, and dependencies among them, it can be represented as a graph. Thus, unlike TF-IDF, PageRank is a natural choice for analyzing the source code. While there have been a few studies [135, 138] along this line, further work is warranted to understand the true potential of PageRank algorithm in various IR-based code retrieval tasks (e.g., code summarization, code reuse).

5.10 Query reformulation as a feasible choice for improved bug localization

Antoniol et al. [9] first use Vector Space Model (VSM) to recover traceability links. Zhou et al. [190] later refine VSM as rVSM and incorporate past bug reports in the IR-based bug localization. Saha et al. [146] make use of structures both from bug reports and from source code documents to localize the bugs. Wong et al. [175] boost up the bug-proneness score of a source document based on the stack traces found in a bug report. Sisman and Kak [158] and Wen et al. [173] incorporate version control history in the IR-based bug localization. Finally, Wang and Lo [169] incorporate five major items – past bug reports, structures, stack traces, version history and author history – from the literature, and outperform the earlier approaches on IR-based localization. Thus, existing literature often adopt an incremental approach and include more and more external artifacts in their approaches. Although these artifacts have positive impact upon the localization performance, their inclusion makes the proposed approaches less scalable and less usable. Besides, these approaches become subject to external dependencies, which is a bad choice from the design point of view. Such limitations might partially explain the reluctance of software practitioners in adopting the IR-based bug localization techniques in their workflow [83, 120, 167, 191].

Alternatively, several studies equipped with query reformulations [31, 56, 135, 138] make an effective use of the primary resources available to the practitioners (e.g., bug report and source code), and then localize the bugs with a competitive accuracy and at low costs. Thus, we believe that query reformulation can be an important part of at least 20 IR-based Software Engineering tasks including bug localization [31], duplicate bug detection [32], bug triaging and bug report summarization. Future work can investigate the impacts of query reformulations upon these 20 tasks and develop more appropriate tools to supports them.

**Summary of RQ6:** Despite significant research, there is still room for further work in automated query reformulations and code searches (e.g., bug localization, concept location, feature location, Internet-scale code search). Existing findings suggest that up to 88% of bug reports contain appropriate keywords that make good queries and can find the bugs using Information Retrieval methods. Unfortunately, many existing techniques are not efficient enough to identify these keywords. The selection of keywords could be further improved by adding more contexts (e.g., time-awareness), designing an appropriate fitness function (for GA-based solutions), leveraging the structures from source code, or using neural language modeling (e.g., word embeddings). The adverse effects of automated query reformulations could be mitigated by combining both human cognitive power and tools’ strengths. Large software repositories (e.g., GitHub) and programming Q&A site (e.g.,
Stack Overflow) have a lot to offer to the research of query reformulations and code search. They store invaluable technical knowledge curated by a large technical crowd, which can be leveraged to support not only code searches but also other software development and maintenance activities.

6 \section*{THREATS TO VALIDITY}

We identify several threats to the validity of our findings in this work. While we did our best to mitigate the majority of these threats, a few of them can be addressed in the future surveys on automated query reformulations. We classify our identified threats into several standard categories and discuss them as follows.

6.1 \section*{Threats to internal validity}

These threats are associated with experimental or internal errors and human biases [184]. Selection of primary studies and their classification into certain categories could be a source of such threats. However, we take careful steps to mitigate them. First, we choose 70 primary studies using a well-established, systematic approach (Section 3) that involves (a) generating keywords from research questions using PIO criterion, (b) extensive searching against 11 widely used publication databases (e.g., IEEE Xplore, ACM Digital library), and (c) multi-level filtration based on specific criteria. We first collect a large set of 2,970 studies that match our search keywords, and then carefully select 70 studies from them using six levels of noise filtration. Second, we use the Grounded Theory approach (Section 3.5) to classify the primary studies based on their adopted methodologies, algorithms (Section 4.1), and experienced challenges (Section 4.3). Grounded Theory has been a popular choice for qualitative analysis in Software Engineering research for decades [31, 51, 161]. Thus, the threats identified above might be mitigated.

Local code search and Internet-scale code search differ in their working context and end goals. While former targets software change tasks (e.g., bug localization, feature location), the latter one focuses on code reuse or general-purpose programming tasks. One might argue about the design of our survey since the query reformulation studies from both searches were captured. However, according to our investigation, query reformulation approaches from these two searches have a significant overlap in their reformulation techniques (Table 4, Fig. 21), evaluation methods (Figures 24, 25), and even strengths or limitations (Fig. 26). Thus, analysis restricted to only either local code searches or Internet-scale code search misses a significant number of strongly related studies from the literature. Besides, collecting both set of studies allows us to compare and contrast between them, which has led us to meaningful insights (Section 4.5) and novel ideas for future work (Section 5). There also exists a large body of work on query reformulations in the Information Retrieval domain [26, 29]. Many of our primary studies adapt their ideas from them. However, since we deal with Software Engineering problems such as code search, those studies from IR literature were not included in our survey.

Our systematic review is restricted to only such studies that deliver either automated or semi-automated tool supports to reformulate queries for code search. That is, it does not include empirical studies, developer surveys and sometimes even the simple incremental works. Such a choice was made to construct a homogeneous set of primary studies that solve a single research problem, i.e., automated query reformulation to support code search.

6.2 \section*{Threats to external validity}

These threats relate to the generalizability of any reported findings [184]. Our primary studies are carefully chosen from a large population of 2,970 results that were retrieved from 11 popular publication databases (Fig. 4). Thus, these primary studies are likely to represent the core studies.
on query reformulation supporting code search in the literature. Consequently, our findings and insights derived from these studies could also generalize for the remaining, accidentally missed studies from the literature. Furthermore, we leverage the Grounded Theory approach for our qualitative analysis, which has been a standard practice for years [31, 33, 51, 161]. Thus, the threats to external validity might be mitigated.

6.3 Threats to construct validity

These threats relate to the appropriateness of evaluation methodology used in a study [184]. Since our selection of primary studies involves certain quality control mechanism, such threats may arise. We select 70 studies through a well-established approach of extensive database search and well-documented filtration criteria (Fig. 4). Furthermore, we control the quality of these studies by asking 10 common questions about their quality (Section 3.4). While the majority of these studies pass the quality check, the remaining few do not meet certain quality standards (e.g., lack of rigorous evaluation and validation). However, we retain them in our analysis due to their strong relevance to our survey topic—query reformulation in source code search. It also should be noted that three studies [40, 42, 50] on traceability link recovery were also included in our survey due to their solid contributions in query reformulation algorithms.

6.4 Threats to conclusion validity

These threats arise when the relationships between two variables are not backed up by solid evidence or reliable data [31]. In our qualitative analysis (Section 4.5), we compare and contrast between local code searches and Internet-scale code search and make several statements about their characteristics, methodologies, strengths, and limitations. We also outline several future research directions based on our analyses. Our claims and suggestions were backed up not only by substantial analytical data (RQ1, RQ3) but also by empirical evidence (RQ4). Thus, threats to the conclusion validity of our systematic survey might also be mitigated.

7 RELATED WORK

Source code search has two major steps—(a) query construction and (b) retrieval of relevant code [56, 162]. There have been several surveys [41, 143, 176] that examine the evidence of existing researches on local code searches (e.g., feature location, bug localization). They mostly focus on the studies that deal with the second step of code search, i.e., retrieval of relevant code. Unfortunately, to the best of our knowledge, there exists no systematic survey that deals with query reformulation approaches adopted in source code search. The closely related work is a similar survey [29] from the Information Retrieval (IR) domain. Although IR-based methods have been widely adopted by at least 20 Software Engineering tasks [56, 115], they experience unique challenges (Section 4.3), which warrants further investigation [116, 167]. Our survey identifies the existing work on query reformulations supporting Software Engineering tasks, categorize them based on their underlying methodologies or limitations, and then discusses their open issues.

Dit et al. [41] first provide a comprehensive survey of 89 studies on feature location techniques (FLT). They analyze each of these studies using seven different dimensions: (a) type of analysis, (b) user input, (c) data sources, (d) output, (e) programming language support, (f) evaluation method, and (g) software systems used. Based on an analysis, they also provide a detailed taxonomy on the FLT studies. Rubin and Chechik [143] perform a similar survey using 24 studies on feature location techniques. They provide necessary guidelines for software practitioners to choose the right feature location technique for their tasks. Wong et al. [176] analyze 331 studies on software fault localization and provide a detailed classification of their studies. Unlike the above work, our survey deals with not only feature location or bug localization but also other types of code
searches such as concept location and Internet-scale code search. More specifically, we deal with a cross-cutting concept namely query reformulation that concerns each of these code search activities. Our survey also analyzes 70 primary studies on query reformulations and categorize them based on several aspects such as query reformulation methodology (\textit{RQ}_1), evaluation method (\textit{RQ}_2), and their experienced challenges or limitations (\textit{RQ}_3).

Zhang et al. [187] present a survey on bug report analysis where they classify the existing work on bug-report optimization (e.g., bug severity prediction), bug-report triage (e.g., bug report assignment), and bug-fixing (e.g., fixing time prediction). Similarly, we also deal with the primary studies that perform bug report analysis. However, our focus was to capture the studies that accept a bug report as an input, construct appropriate queries from the input, and then localize the bug using these queries.

A few other studies [31, 32, 95, 115] also investigate the existing work on query reformulations supporting code searches and other Software Engineering tasks. However, their analysis was limited and restricted to either local [31, 115] or Internet-scale code search [95]. Many of them also do not apply a systematic approach [80] for study selection, quality control, and qualitative analysis. Unlike them, we conduct a systematic literature review by (a) asking important research questions, (b) carefully selecting the primary studies from 11 popular publication databases, and (c) performing qualitative analysis with the Grounded Theory approach [51], a widely used qualitative technique. No earlier survey addresses the same topic in this scale, which makes our work novel.

8 CONCLUSION

Software maintenance can cost up to 80% of total budget in modern software development. Searching for the code of interest (e.g., buggy code, reusable code example) is an integral part of software maintenance. Developers often choose a few important keywords capturing their information need and attempt to find the code of interest either from a local software codebase or an Internet-scale code repository (e.g., GitHub). Unfortunately, existing evidence suggests that the developers often perform poorly in choosing their keywords. As a result, they spend a significant amount of their time manually reformulating queries and analyzing the corresponding search results. Towards this goal, a number of studies attempt to support the developers through automated query reformulations. In this article, we conduct a systematic literature review using 70 primary studies on automated query reformulations intended for source code search. We summarize our quantitative and qualitative findings below and answer our research questions briefly as follows.

- \textbf{RQ}_1: Existing approaches for query reformulation adopt eight major methodologies including term weighting, relevance feedback, semantic relations, thesaurus lookup, and data mining to reformulate their queries and to support code searches in various working contexts.
- \textbf{RQ}_2: Existing studies evaluate their approaches with empirical methods, developer surveys, or using both. They use a total of 30 performance metrics including the popular ones (e.g., Hit@K, MAP). However, developer involvement is generally low, and the number of queries used to evaluate queries in the Internet-scale code search is often small. About 30% of the primary studies were selected for comparison by the later studies.
- \textbf{RQ}_3: Existing query reformulation approaches suffer from eight major challenges and limitations including noisy keywords in their queries, vocabulary mismatch problem, lack of generalizability, human-induced biases, weak evaluation, and other prevailing issues that might prevent them from adoption by the software practitioners.
- \textbf{RQ}_4: A total of 70 primary studies were conducted over the last 15 years where 80% of them were done in the last nine years. These studies appeared in the top Software Engineering venues such as ICSE, ASE, ICSME, TSE, TOSEM, and EMSE. Given the increasing size
and complexity of modern software systems, the research topics such as automated query reformulation and code search are likely to gain more popularity in the coming years.

- **RQ5**: About 58% of our primary studies perform query reformulations to support local code searches (e.g., concept location, bug localization, feature location) whereas the remaining 42% focus on Internet-scale code search. The primary studies from these two searches differ in their adopted methodologies, reformulation types, evaluation methods, and human participation. However, both groups of studies suffer from a common set of issues such as a lack of generalizability, weak evaluation, and noisy queries.

- **RQ6**: Constructing the right queries for code search has been a challenge for decades. Despite significant research, there is still room for further investigation. Future work can focus on adding more contexts (e.g., time-awareness) to their keyword selection algorithms, designing an appropriate fitness function for GA-based solutions, leveraging the structures from source code, or using neural language modeling (e.g., word embeddings) to deliver better queries for the code search.

**ACKNOWLEDGEMENT**

This research was supported by Tenure-track startup grant, Dalhousie University, International Dean’s Scholarship from University of Saskatchewan (2014–2017), Saskatchewan Innovation & Opportunity Scholarship (2017–2018), and the Natural Sciences and Engineering Research Council of Canada (NSERC).
## SELECTED PRIMARY STUDIES

Table 14 shows the 70 primary studies selected for our systematic literature review.

| ID | Primary Study | Description |
|----|---------------|-------------|
| S1 | Zamani et al. [185] | Query reduction using noun term selection and time-aware term weighting for concept location |
| S2 | Satter and Sakib [150] | Query expansion using keyword co-occurrences across the past queries from code search logs |
| S3 | Lemos et al. [88] | Ad hoc keyword selection from given test cases for semantic code search |
| S4 | Wang et al. [170] | Query expansion using modified Rocchio’s method and the re-ranking of results for improved code search |
| S5 | Yu et al. [183] | Query reduction by extracting nouns, verbs, adjectives and type information from a free-form query for code search |
| S6 | Sisman and Kak [159] | Query reformulation using spatial code proximity for Information Retrieval-based bug localization |
| S7 | Sirres et al. [156] | Query expansion using structured code entities from the relevant Q & A threads of Stack Overflow |
| S8 | Haiduc et al. [57] | Automatic performance prediction of a given search query using query difficulty analysis and machine learning without executing the query |
| S9 | Perez et al. [121] | Query reformulation for feature location using Rocchio, RSV and Dice methods with software model families |
| S10 | Haiduc et al. [56] | Best query reformulation suggestion using Rocchio, RSV, Dice methods, query difficulty analysis and machine learning |
| S11 | Kevic and Fritz [75] | Query reduction using advanced heuristics and logistic regression |
| S12 | Hill et al. [62] | Semi-automated reformulation of NL queries with query keyword contexts and NL phrase extraction from method and field signatures |
| S13 | Lemos et al. [89] | Query reformulation with NL thesaurus (e.g., WordNet) in the context of interface-driven and keyword-based code searches |
| S14 | Martie et al. [107] | Query reformulation using frequent identifier names from result files and other contextual information |
| S15 | Lv et al. [102] | Query reformulation with relevant APIs by matching query texts and corresponding API documentations |
| S16 | Li et al. [92] | Query expansion by using tag co-occurrences in the same and duplicate questions of Stack Overflow |
| S17 | Ge et al. [48] | Query reformulation using pre-search (e.g., identifier and Verb-DO completion, term co-occurrences) and post-search recommendations (e.g., synonym suggestion, spelling corrections) |
| S18 | Dourdas et al. [42] | Query reformulation using ontology models for web service discovery |
| S19 | Rahman and Roy [137] | Query reformulation using crowd generated knowledge and large-scale data analytics derived from Stack Overflow corpus |
| S20 | Zhang et al. [186] | Query reformulation using semantically related API classes for the improved Internet-scale code search |
| S21 | Sisman et al. [157] | Query reformulation using spatial code proximity, term ordering, Markov Random Field and query conditioning |
| S22 | Rahman and Roy [135] | Query reformulation using term weighting method, CodeRank, query difficulty analysis, and machine learning |
| S23 | Lapeña et al. [84] | Query reformulation of NL query using textual similarity with product requirements for feature location |

Continued on next page
| ID  | Primary Study                        | Description                                                                                                                                 |
|-----|--------------------------------------|---------------------------------------------------------------------------------------------------------------------------------------------|
| S24 | Wang et al. [166]                    | Semi-automated query reformulation using structure, intent and dependency facets of search results                                           |
| S25 | Rahman and Roy [138]                 | Query reformulation using bug report quality dynamics and graph-based term weighting algorithms                                             |
| S26 | Lin et al. [93]                      | Query reformulation using relevant API entities from RecoDec, conceptual knowledge, TF-IDF, and advanced heuristics                           |
| S27 | Yang and Tan [180]                   | Query reformulation using semantically similar word pairs mined from comment-code contexts of source documents                               |
| S28 | Howard et al. [65]                   | Query reformulation using semantically similar word pairs mined from leading comments and method signatures across projects                |
| S29 | Chaparro and Marcus [34]             | Query reduction using gradual term removal with the help of query effectiveness                                                              |
| S30 | Gay et al. [47]                      | Query reformulation using explicit relevance feedback from the developers and Rocchio’s expansion method                                      |
| S31 | Panchenko et al. [119]               | Query reformulation with AST generation and XPath representation of example code snippet                                                    |
| S32 | Mills et al. [114]                   | Best query prediction using query difficulty analysis and machine learning for concept location and traceability recovery                    |
| S33 | Balachandran [15]                    | Query reduction by subtree generation from the AST of input code                                                                           |
| S34 | Nie et al. [118]                     | Query expansion using Rocchio’s method and pseudo-relevance feedback from Stack Overflow                                                   |
| S35 | Huang et al. [69]                    | Query expansion based on intent mining from code change history and machine learning                                                         |
| S36 | Lu et al. [99]                       | Query expansion using POS tagging and synonym addition from WordNet for improved concept location                                            |
| S37 | Li et al. [91]                       | Query reformulation by term replacement with synonymous tags and by term expansion with related tags from Stack Overflow                     |
| S38 | Rahman and Roy [134]                 | Query reformulation using frequently co-occurred keywords from source code and Stack Overflow threads                                         |
| S39 | Rahman et al. [139]                  | Query reformulation using keyword-API co-occurrences in the Q & A threads of Stack Overflow                                                   |
| S40 | Rahman and Roy [136]                 | Query reduction by selecting important keywords using graph-based term weighting namely PageRank                                               |
| S41 | Raghothaman et al. [127]             | Query reformulation using co-occurrence probability of keyword and API classes in the search engine logs                                   |
| S42 | Lemos et al. [90]                    | Query expansion using synonyms/antonyms from three thesauri– WordNet, Yang and Tan [180], and type thesaurus                                 |
| S43 | Shepherd et al. [153]                | Query reformulation using Verb-DO extraction and their expansion using stemmed words, synonyms and similar usages                             |
| S44 | Chaparro et al. [31]                 | Query reduction using observed behaviour identification, Grounded Theory and natural language discourse analysis                            |
| S45 | Kevic and Fritz [74]                 | Query reformulation using NL-SCL mapping between changeset and IDE interaction history, and term weighting                                    |
| S46 | Gvero and Kuncak [53]                | Query reformulation with API method signatures using sentence parse tree extraction, uni-gram, and probabilistic context free grammar (PCFG) |
| S47 | Kimmig et al. [79]                   | Query reduction using POS tagging, selection of nouns and verbs and source code ontology                                                    |

Continued on next page
| ID | Primary Study          | Description                                                                                                                                                                                                 |
|----|------------------------|-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| S48 | Wursch et al. [178]    | Query reformulation using semantic web based ontology models, Ginseng, and SPARQL                                                                                                                           |
| S49 | Vinayakarao et al. [165] | Query/corpus reformulation using the mapping between programming concepts and their syntactic forms from Stack Overflow Q&A threads                                                                            |
| S50 | Marcus et al. [106]     | Query reformulation using similar terms from the source code with Latent Semantic Indexing (LSI)                                                                                                           |
| S51 | Gibiec et al. [50]      | Query reformulation using web data mining and term weighting methods—domain term frequency, domain specificity, and concept generality                                                                          |
| S52 | Dietrich et al. [40]    | Query reformulation using trace query transformation rules generated by FP-growth algorithm                                                                                                               |
| S53 | Jiang et al. [71]       | Query reformulation using ontology models across multiple repositories and candidate navigation path ranking                                                                                                 |
| S54 | Bassett and Kraft [16]  | Query/corpus reformulation by weight boosting (e.g., repeating) of structured entities from various contexts in the source code                                                                              |
| S55 | Chatterjee et al. [35]  | Query/corpus expansion by appending corresponding API documentations against method invocations.                                                                                                          |
| S56 | Wang et al. [171]       | Query suggestion by mining program dependency graphs from a given code segment                                                                                                                               |
| S57 | Zhang et al. [188]      | Query reformulation by expanding methods with neighboring methods based on semantic similarity, temporal proximity, and call dependency                                                                     |
| S58 | Liu et al. [97]         | Query expansion using co-occurring API methods from the underlying corpus using encoder-decoder transformation and conditional probability analysis                                                         |
| S59 | Pérez et al. [122]      | Query expansion using experts’ descriptions of a feature, their confidence level, and Rocchio’s method                                                                                                      |
| S60 | Lu et al. [98]          | Query expansion using first and second order word co-occurrences, and object-oriented relations among the words                                                                                              |
| S61 | Kim and Lee [77]        | Query reformulation by adding attachment contents and by reducing noisy keywords from a bug report                                                                                                             |
| S62 | Kim and Lee [78]        | Query reformulation using Genetic algorithm and multi-objective optimization (e.g., query difficulty, OB, S2E, EB)                                                                                             |
| S63 | Huang et al. [67]       | Query expansion by deep learning change sequences and predicting the potentially changed terms for improved code search                                                                                      |
| S64 | Pérez et al. [124]      | Query reformulation for feature location using Rocchio’s method, relevance feedback, and expert’s descriptions of a feature as the feedback documents                                                         |
| S65 | Wu and Yang [177]       | Query reformulation using old code examples as relevance feedback and then extracting expansion keywords (i.e., intent) from the change history of those code examples                                                   |
| S66 | Cao et al. [25]         | Query reformulation by learning to predict reformulated query from search logs of Stack Overflow using sequence-to-sequence learning and attention mechanism                                                       |
| S67 | Lawrie and Binkley [86] | Query selection for bug localization using Genetic algorithm and information need analysis as a fitness function                                                                                             |
| S68 | Mills et al. [115]      | Query reformulation for bug localization using Genetic algorithm and query effectiveness as a fitness function                                                                                               |

Continued on next page
Table 14 – Continued from previous page

| ID | Primary Study                  | Description                                                                                                                                 |
|----|--------------------------------|---------------------------------------------------------------------------------------------------------------------------------------------|
| S69 | Pérez et al. [125]           | Query reformulation for feature location using Genetic algorithm using expansion, reduction, and selection as a mutation operation, and textual similarity as a fitness function |
| S70 | Da Silva et al. [38]         | Query expansion for code example search using semantic similarity analysis, API suggestion, and term weighting (e.g., TF-IDF)                  |
### B QUALITY ASSESSMENT OF PRIMARY STUDIES

Table 15 shows answers for the questions outlined in Section 3.4. We represent our responses as follows—● as "Yes", ○ as "No", and ◀ or ◁ as "Somewhat Yes".

| ID | Primary Study     | Q1 | Q2 | Q3 | Q4 | Q5 | Q6 | Q7 | Q8 | Q9 | Q10 |
|----|-------------------|----|----|----|----|----|----|----|----|----|-----|
| S1 | Zamani et al. [185] | ●  | ●  | ●  | ●  | ●  | ●  | ●  | ●  | ●  | ★   |
| S2 | Satter and Sakib [150] | ●  | ●  | ●  | ●  | ●  | ●  | ○  | ○  | ○  | ★   |
| S3 | Lemos et al. [88]   | ●  | ●  | ●  | ●  | ●  | ●  | ○  | ●  | ●  | ★   |
| S4 | Wang et al. [170]   | ●  | ●  | ●  | ●  | ●  | ●  | ●  | ●  | ●  | ○   |
| S5 | Yu et al. [183]     | ●  | ●  | ●  | ●  | ●  | ●  | ○  | ○  | ◁  | ○   |
| S6 | Sisman and Kak [159] | ●  | ●  | ●  | ●  | ●  | ●  | ●  | ●  | ●  | ◀   |
| S7 | Sirres et al. [156] | ●  | ●  | ●  | ●  | ●  | ●  | ●  | ●  | ●  | ★   |
| S8 | Haiduc et al. [57]  | ●  | ●  | ○  | ●  | ●  | ●  | ●  | ●  | ●  | ○   |
| S9 | Perez et al. [121]  | ●  | ●  | ●  | ●  | ●  | ●  | ○  | ○  | ○  | ○   |
| S10| Haiduc et al. [56]  | ●  | ●  | ●  | ●  | ●  | ●  | ●  | ●  | ●  | ○   |
| S11| Kevic and Fritz [75] | ●  | ●  | ●  | ●  | ●  | ●  | ●  | ●  | ●  | ○   |
| S12| Hill et al. [62]    | ●  | ●  | ●  | ●  | ●  | ●  | ●  | ●  | ●  | ●   |
| S13| Lemos et al. [89]   | ●  | ●  | ●  | ●  | ●  | ●  | ●  | ●  | ●  | ○   |
| S14| Martie et al. [107] | ●  | ●  | ●  | ●  | ●  | ●  | ●  | ●  | ●  | ○   |
| S15| Lv et al. [102]     | ●  | ●  | ●  | ●  | ●  | ●  | ●  | ●  | ●  | ●   |
| S16| Li et al. [92]      | ●  | ●  | ●  | ●  | ●  | ●  | ●  | ●  | ●  | ○   |
| S17| Ge et al. [48]      | ●  | ●  | ●  | ●  | ●  | ●  | ●  | ●  | ●  | ●   |
| S18| Dourdas et al. [42] | ●  | ●  | ●  | ●  | ●  | ●  | ●  | ●  | ●  | ○   |
| S19| Rahman and Roy [137]| ●  | ●  | ●  | ●  | ●  | ●  | ●  | ●  | ●  | ●   |
| S20| Zhang et al. [186]  | ●  | ●  | ●  | ●  | ●  | ●  | ●  | ●  | ●  | ○   |
| S21| Sisman et al. [157] | ●  | ●  | ●  | ●  | ●  | ●  | ●  | ●  | ●  | ○   |
| S22| Rahman and Roy [135]| ●  | ●  | ●  | ●  | ●  | ●  | ●  | ●  | ●  | ○   |
| S23| Lapeña et al. [84]  | ●  | ●  | ●  | ●  | ●  | ●  | ●  | ●  | ●  | ○   |
| S24| Wang et al. [166]   | ●  | ●  | ○  | ●  | ●  | ●  | ●  | ●  | ●  | ●   |
| S25| Rahman and Roy [138]| ●  | ●  | ●  | ●  | ●  | ●  | ●  | ●  | ●  | ●   |
| S26| Lin et al. [93]     | ●  | ●  | ●  | ●  | ●  | ●  | ●  | ●  | ●  | ●   |
| S27| Yang and Tan [180]  | ●  | ●  | ●  | ●  | ●  | ●  | ●  | ●  | ●  | ●   |
| S28| Howard et al. [65]  | ●  | ●  | ●  | ●  | ●  | ●  | ●  | ●  | ●  | ●   |
| S29| Chaparro and Marcus [34]| ●  | ○  | ○  | ●  | ●  | ●  | ●  | ●  | ●  | ●   |
| S30| Gay et al. [47]     | ●  | ●  | ●  | ●  | ●  | ●  | ●  | ●  | ●  | ●   |
| S31| Panchenko et al. [119]| ●  | ●  | ●  | ●  | ●  | ●  | ●  | ●  | ●  | ●   |
| S32| Mills et al. [114]  | ●  | ●  | ●  | ●  | ●  | ●  | ●  | ●  | ●  | ●   |
| S33| Balachandran [15]   | ●  | ●  | ●  | ●  | ●  | ●  | ●  | ●  | ●  | ●   |
| S34| Nie et al. [118]    | ●  | ●  | ●  | ●  | ●  | ●  | ●  | ●  | ●  | ●   |
| S35| Huang et al. [69]   | ●  | ●  | ●  | ●  | ●  | ●  | ●  | ●  | ●  | ●   |
| S36| Lu et al. [99]      | ●  | ●  | ●  | ●  | ●  | ●  | ●  | ●  | ●  | ●   |
| S37| Li et al. [91]      | ●  | ●  | ●  | ●  | ●  | ●  | ●  | ●  | ●  | ●   |
| S38| Rahman and Roy [134]| ●  | ●  | ●  | ●  | ●  | ●  | ●  | ●  | ●  | ●   |
| S39| Rahman et al. [139] | ●  | ●  | ●  | ●  | ●  | ●  | ●  | ●  | ●  | ●   |
| S40| Rahman and Roy [136]| ●  | ●  | ●  | ●  | ●  | ●  | ●  | ●  | ●  | ●   |
| S41| Raghothaman et al. [127]| ●  | ●  | ●  | ●  | ●  | ●  | ●  | ●  | ●  | ●   |

Continued on next page
| ID  | Primary Study            | Q1 | Q2 | Q3 | Q4 | Q5 | Q6 | Q7 | Q8 | Q9 | Q10 |
|-----|-------------------------|----|----|----|----|----|----|----|----|----|-----|
| S42 | Lemos et al. [90]       | ●  | ●  | ●  | ●  | ●  | ●  | ●  | ●  | ●  | ●   |
| S43 | Shepherd et al. [153]   | ●  | ●  | ●  | ●  | ●  | ●  | ●  | ●  | ●  | □   |
| S44 | Chaparro et al. [31]    | ●  | ●  | ●  | ●  | ●  | ●  | ●  | ●  | ●  | ●   |
| S45 | Kevic and Fritz [74]    | ●  | ●  | ●  | ●  | ●  | ●  | ●  | ●  | ●  | ●   |
| S46 | Gvero and Kuncak [53]   | ●  | ●  | ●  | ●  | ●  | ●  | ●  | ●  | ●  | □   |
| S47 | Kimmig et al. [79]      | ●  | ●  | ●  | ●  | ●  | ●  | ●  | ●  | ●  | ●   |
| S48 | Wursch et al. [178]     | ●  | ●  | ●  | ●  | ●  | ●  | ●  | ●  | ●  | □   |
| S49 | Vinayakara et al. [165] | ●  | ●  | ●  | ●  | ●  | ●  | ●  | ●  | ●  | ●   |
| S50 | Marcus et al. [106]     | ●  | ●  | ●  | ●  | ●  | ●  | ●  | ●  | ●  | □   |
| S51 | Gibiec et al. [50]      | ●  | ●  | ●  | ●  | ●  | ●  | ●  | ●  | ●  | ●   |
| S52 | Dietrich et al. [40]    | ●  | ●  | ●  | ●  | ●  | ●  | ●  | ●  | ●  | □   |
| S53 | Jiang et al. [71]       | ●  | ●  | ●  | ●  | ●  | ●  | ●  | ●  | ●  | □   |
| S54 | Eddy et al. [43]        | ●  | ●  | ●  | ●  | ●  | ●  | ●  | ●  | ●  | □   |
| S55 | Chatterjee et al. [35]  | ●  | ●  | ●  | ●  | ●  | ●  | ●  | ●  | ●  | □   |
| S56 | Wang et al. [171]       | ●  | ●  | ●  | ●  | ●  | ●  | ●  | ●  | ●  | □   |
| S57 | Zhang et al. [188]      | ●  | ●  | ●  | ●  | ●  | ●  | ●  | ●  | ●  | □   |
| S58 | Liu et al. [97]         | ●  | ●  | ●  | ●  | ●  | ●  | ●  | ●  | ●  | □   |
| S59 | Pérez et al. [122]      | ●  | ●  | ●  | ●  | ●  | ●  | ●  | ●  | ●  | □   |
| S60 | Lu et al. [98]          | ●  | ●  | ●  | ●  | ●  | ●  | ●  | ●  | ●  | □   |
| S61 | Kim and Lee [77]        | ●  | ●  | ●  | ●  | ●  | ●  | ●  | ●  | ●  | □   |
| S62 | Kim and Lee [78]        | ●  | ●  | ●  | ●  | ●  | ●  | ●  | ●  | ●  | □   |
| S63 | Huang et al. [67]       | ●  | ●  | ●  | ●  | ●  | ●  | ●  | ●  | ●  | □   |
| S64 | Pérez et al. [124]      | ●  | ●  | ●  | ●  | ●  | ●  | ●  | ●  | ●  | □   |
| S65 | Wu and Yang [177]       | ●  | ●  | ●  | ●  | ●  | ●  | ●  | ●  | ●  | □   |
| S66 | Cao et al. [25]         | ●  | ●  | ●  | ●  | ●  | ●  | ●  | ●  | ●  | □   |
| S67 | Lawrie and Binkley [86] | ●  | ●  | ●  | ●  | ●  | ●  | ●  | ●  | ●  | □   |
| S68 | Mills et al. [115]      | ●  | ●  | ●  | ●  | ●  | ●  | ●  | ●  | ●  | □   |
| S69 | Pérez et al. [125]      | ●  | ●  | ●  | ●  | ●  | ●  | ●  | ●  | ●  | □   |
| S70 | Da Silva et al. [38]    | ●  | ●  | ●  | ●  | ●  | ●  | ●  | ●  | ●  | □   |
REFERENCES

[1] 2018. Boeing eyes Lion Air crash software upgrade in 6 to 8 weeks. (2018). https://goo.gl/Xy1qFa

[2] 2019. The 737Max and Why Software Engineers Might Want to Pay Attention. (2019). https://bit.ly/2CmeTqB

[3] 2019. Apache Lucene Core. (2019). https://lucene.apache.org/core

[4] 2019. Boeing 737 jets grounded globally as officials investigate technical issues behind fatal crash. (2019). https://goo.gl/ieBgYN

[5] 2019. Codota Code Search. (2019). https://www.codota.com/code

[6] 2019. GitHub Code Search. (2019). https://github.com/search

[7] 2019. Here’s the terrifying reason Boeing’s 737 MAX 8 is grounded across the globe. (2019). https://goo.gl/GwXv6H

[8] 2021. Replication package: A Systematic Literature Review of Automated Query Reformulations in Source Code Search. (2021). https://bit.ly/3eccmlZ

[9] G Antoniol, G Canfora, G Casazza, A De Lucia, and E Merlo. 2002. Recovering Traceability Links between Code and Documentation. TSE 28, 10 (2002), 970–983.

[10] J. Anvik, L. Hiew, and G. C. Murphy. 2005. Coping with an Open Bug Repository. In Proc. OOPSLA/Eclipse. 35–39.

[11] M. Asaduzzaman, C. K. Roy, K. A. Schneider, and D. Hou. 2016. A Simple, Efficient, Context-Sensitive Approach for Code Completion. JSEP 28, 7 (2016), 512–541.

[12] S. Bajracharya and C. Lopes. 2009. Mining search topics from a code search engine usage log. In Proc. MSR. 111–120.

[13] S. Bajracharya, T. Ngo, E. Linstead, Y. Dou, P. Rigor, P. Baldi, and C. Lopes. 2006. Sourcerer: A Search Engine for Open Source Code Supporting Structure-based Search. In Proc. OOPSLA-C. 681–682.

[14] S. K. Bajracharya and C. V. Lopes. 2012. Analyzing and Mining a Code Search Engine Usage Log. EMSE 17, 4-5 (2012), 424–466.

[15] V. Balachandran. 2015. Query by example in large-scale code repositories. In Proc. SANER. 467–476.

[16] B Bassett and N A Kraft. 2013. Structural Information based Term Weighting in Text Retrieval for Feature Location. In Proc. ICPC. 133–141.

[17] R Blanco and C Lioma. 2012. Graph-based Term Weighting for Information Retrieval. Inf. Retr. 15, 1 (2012), 54–92.

[18] D. M. Blei, A. Y. Ng, and M. I. Jordan. 2003. Latent Dirichlet Allocation. J. Mach. Learn. Res. 3 (2003), 993–1022.

[19] P. Bojanowski, E. Grave, A. Joulin, and T. Mikolov. 2016. Enriching Word Vectors with Subword Information. arXiv preprint arXiv:1607.04606 (2016).

[20] A. Bosu. 2014. Characteristics of the Vulnerable Code Changes Identified Through Peer Code Review. In Proc. ICSE-C (ICSE Companion 2014). 736–738.

[21] A. Bosu, J. C. Carver, M. Hafiz, P. Hilley, and D. Janni. 2014. Identifying the Characteristics of Vulnerable Code Changes: An Empirical Study. In Proc. FSE. 257–268.

[22] J Brandt, P J Guo, J Lewenstein, M Dontcheva, and S R Klemmer. 2009. Two Studies of Opportunistic Programming: Interleaving Web Foraging, Learning, and Writing Code. In Proc. SIGCHI. 1589–1598.

[23] S Brin and L Page. 1998. The Anatomy of a Large-Scale Hypertextual Web Search Engine. Comput. Netw. ISDN Syst. 30, 1-7 (1998), 107–117.
[24] D. Cai, C. J. van Rijsbergen, and J. M. Jose. 2001. Automatic Query Expansion Based on Divergence. In Proc. CIKM. 419–426.

[25] K. Cao, C. Chen, S. Baltes, C. Treude, and X. Chen. 2021. Automated Query Reformulation for Efficient Search based on Query Logs From Stack Overflow. In Proc. ICSE. 13.

[26] D Carmel and E Yom-Tov. 2010. Estimating the Query Difficulty for Information Retrieval. Morgan & Claypool. 89– pages.

[27] D Carmel, E Yom-Tov, A Darlow, and D Pelleg. 2006. What Makes a Query Difficult?. In Proc. SIGIR. 390–397.

[28] C. Carpineto, R. de Mori, G. Romano, and B. Bigi. 2001. An Information-theoretic Approach to Automatic Query Expansion. ACM Trans. Inf. Syst. 19, 1 (2001), 1–27.

[29] C Carpineto and G Romano. 2012. A Survey of Automatic Query Expansion in Information Retrieval. ACM Comput. Surv. 44, 1 (2012), 1:1–1:50.

[30] W Chan, H Cheng, and D Lo. 2012. Searching Connected API Subgraph via Text Phrases. In Proc. FSE. 10:1–10:11.

[31] O Chaparro, J M Florez, and A Marcus. 2017. Using Observed Behavior to Reformulate Queries during Text Retrieval-based Bug Localization. In Proc. ICSME. 376–387.

[32] O. Chaparro, J. M. Florez, U. Singh, and A. Marcus. 2019. Reformulating Queries for Duplicate Bug Report Detection. In Proc. SANER. 12.

[33] O Chaparro, J Lu, F Zampetti, L Moreno, M Di Penta, A Marcus, G Bavota, and V Ng. 2017. Detecting Missing Information in Bug Descriptions. In Proc. ESEC/FSE. 396–407.

[34] O Chaparro and A Marcus. 2016. On the Reduction of Verbose Queries in Text Retrieval Based Software Maintenance. In Proc. ICSE-C. 716–718.

[35] S. Chatterjee, S. Juvekar, and K. Sen. 2009. SNIFF: A Search Engine for Java Using Free-Form Queries. In Proc. FSE. 385–400.

[36] C. Chen, Z. Xing, and Y. Liu. 2019. What’s Spain’s Paris? Mining Analogical Libraries from Q&A Discussions. EMSE 24, 3 (2019), 1155–1194.

[37] J Cordeiro, B Antunes, and P Gomes. 2012. Context-based Recommendation to Support Problem Solving in Software Development. In Proc. RSSE. 85–89.

[38] R. F. G. Da Silva, C. K. Roy, M. M. Rahman, K. Schneider, K. Paixo, and M. Maia. 2019. Recommending Comprehensive Solutions for Programming Tasks by Mining Crowd Knowledge. In Proc. ICPC. 358–368.

[39] B. Dagenais and M. P. Robillard. 2012. Recovering traceability links between an API and its learning resources. In Proc. ICSE. 47–57.

[40] T. Dietrich, J. Cleland-Huang, and Y. Shin. 2013. Learning effective query transformations for enhanced requirements trace retrieval. In Proc. ASE. 586–591.

[41] B. Dit, M. Revelle, M. Gethers, and D. Poshivanyk. [n. d.]. Feature location in source code: a taxonomy and survey. JSEP 25, 1 ([n. d.]), 53–95.

[42] N. Dourdas, X. Zhu, N. Maiden, S. Jones, and K. Zachos. 2006. Discovering Remote Software Services that Satisfy Requirements: Patterns for Query Reformulation. In Advanced Information Systems Engineering. 239–254.

[43] Brian P. Eddy, Nicholas A. Kraft, and Jeff Gray. 2018. Impact of structural weighting on a latent Dirichlet allocation based feature location technique. JSEP 30, 1 (2018), e1892.

[44] F. Ensan, E. Bagheri, and M. Kahani. 2007. The Application of Users’ Collective Experience for Crafting Suitable Search Engine Query Recommendations. In Proc. CNSR. 148–156.

[45] L Favre. 2008. Modernizing Software & System Engineering Processes. In Proc. ICSENG. 442–447.

[46] G W Furnas, T K Landauer, L M Gomez, and S T Dumais. 1987. The Vocabulary Problem in Human-system Communication. Commun. ACM 30, 11 (1987), 964–971.
[47] G Gay, S Haiduc, A Marcus, and T Menzies. 2009. On the Use of Relevance Feedback in IR-based Concept Location. In Proc. ICSM. 351–360.

[48] X. Ge, D. C. Shepherd, K. Damevski, and E. Murphy-Hill. 2017. Design and evaluation of a multi-recommendation system for local code search. Journal of Visual Languages and Computing 39 (2017), 1 – 9.

[49] M. Ghafari and H. Moradi. 2017. A framework for classifying and comparing source code recommendation systems. In Proc. SANER. 555–556.

[50] M. Gibiec, A. Czauderna, and J. Cleland-Huang. 2010. Towards Mining Replacement Queries for Hard-to-retrieve Traces. In Proc. ASE. 245–254.

[51] B. G. Glaser and A. L. Strauss. 1967. The discovery of grounded theory: strategies for qualitative research. Chicago: Aldine Publishing.

[52] R. L. Glass. 2001. Frequently forgotten fundamental facts about software engineering. IEEE Software 18, 3 (2001), 112–111.

[53] T Gvero and V Kuncak. 2015. Interactive Synthesis Using Free-form Queries. In Proc. ICSE. 689–692.

[54] S Haiduc. 2011. Automatically Detecting the Quality of the Query and its Implications in IR-based Concept Location. In Proc. ASE. 637–640.

[55] Sonia Haiduc. 2013. Supporting Text Retrieval Query Formulation In Software Engineering. PhD dissertation. Wayne State University.

[56] S Haiduc, G Bavota, A Marcus, R Oliveto, A De Lucia, and T Menzies. 2013. Automatic Query Reformulations for Text Retrieval in Software Engineering. In Proc. ICSE. 842–851.

[57] S Haiduc, G Bavota, R Oliveto, A De Lucia, and A Marcus. 2012. Automatic Query Performance Assessment During the Retrieval of Software Artifacts. In Proc. ASE. 90–99.

[58] S Haiduc and A Marcus. 2011. On the Effect of the Query in IR-based Concept Location. In Proc. ICPC. 234–237.

[59] J. H. Hayes, A. Dekhtyar, and S. K. Sundaram. 2006. Advancing candidate link generation for requirements tracing: the study of methods. TSE 32, 1 (2006), 4–19.

[60] V. J. Hellendoorn and P. Devanbu. 2017. Are Deep Neural Networks the Best Choice for Modeling Source Code?. In Proc. ESEC/FSE. 763–773.

[61] Emily Hill. 2010. Integrating natural language and program structure information to improve software search and exploration. PhD dissertation. University of Delaware.

[62] E Hill, L Pollock, and K Vijay-Shanker. 2009. Automatically Capturing Source Code Context of NL-queries for Software Maintenance and Reuse. In Proc. ICSE. 232–242.

[63] E Hill, L Pollock, and K Vijay-Shanker. 2011. Improving Source Code Search with Natural Language Phrasal Representations of Method Signatures. In Proc. ASE. 524–527.

[64] R Holmes and G C Murphy. 2005. Using Structural Context to Recommend Source Code Examples. In Proc. ICSE. 117–125.

[65] M J Howard, S Gupta, L Pollock, and K Vijay-Shanker. 2013. Automatically Mining Software-based, Semantically-Similar Words from Comment-Code Mappings. In Proc. MSR. 377–386.

[66] Q. Huang, X. Xia, Z. Xing, D. Lo, and X. Wang. 2018. API Method Recommendation without Worrying about the Task-API Knowledge Gap. In Proc. ASE (ASE 2018). 293–304.

[67] Q. Huang, Y. Yang, and M. Cheng. 2019. Deep learning the semantics of change sequences for query expansion. SPE 49, 11 (2019), 1600–1617.

[68] Q. Huang, Y. Yang, X. Wang, H. Wan, R. Wang, and G. Wu. 2017. Query Expansion via Intent Predicting. IJSEKE 27, 09n10 (2017), 1591–1601.

[69] Q. Huang, Y. Yang, X. Zhan, H. Wan, and G. Wu. 2018. Query expansion based on statistical learning from code changes. SPE 48, 7 (2018), 1333–1351.
[70] S. F. Hussain and G. Bisson. 2010. *Text Categorization Using Word Similarities Based on Higher Order Co-occurrences*. 1–12.
[71] S. Jiang, L. Shen, X. Peng, Z. Lv, and W. Zhao. 2015. Understanding developers’ natural language queries with interactive clarification. In *Proc. SANER*. 13–22.
[72] K S Jones. 1972. A Statistical Interpretation Of Term Specificity And Its Application In Retrieval. *Journal of Documentation* 28, 1 (1972), 11–21.
[73] D. Kelly and J. Teevan. 2003. Implicit Feedback for Inferring User Preference: A Bibliography. *SIGIR Forum* 37, 2 (2003), 18–28.
[74] K Kevic and T Fritz. 2014. A Dictionary to Translate Change Tasks to Source Code. In *Proc. MSR*. 320–323.
[75] K Kevic and T Fritz. 2014. Automatic Search Term Identification for Change Tasks. In *Proc. ICSE*. 468–471.
[76] M. Kim and E. Lee. 2018. Are Information Retrieval-based Bug Localization Techniques Trustworthy??. In *Proc. ICSE*. 248–249.
[77] M. Kim and E. Lee. 2019. A Novel Approach to Automatic Query Reformulation for IR-Based Bug Localization. In *Proc. SAC*. 1752–1759.
[78] M. Kim and E. Lee. 2020. ManQ: Many-objective optimization-based automatic query reduction for IR-based bug localization. *IST* 125 (2020).
[79] M. Kimmig, M. Monperrus, and M. Mezini. 2011. Querying source code with natural language. In *Proc. ASE*. 376–379.
[80] B. Kitchenham and P. Brereton. 2013. A Systematic Review of Systematic Review Process Research in Software Engineering. *IST* 55, 12 (2013), 2049–2075.
[81] B. Kitchenham and S Charters. 2007. Guidelines for performing Systematic Literature Reviews in Software Engineering. (2007).
[82] Andrew J. Ko, Brad A. Myers, Michael J. Coblenz, and Htet Htet Aung. 2006. An Exploratory Study of How Developers Seek, Relate, and Collect Relevant Information During Software Maintenance Tasks. *TSE* 32, 12 (2006), 971–987.
[83] P. S. Kochhar, Y. Tian, and D. Lo. 2014. Potential Biases in Bug Localization: Do They Matter??. In *Proc. ASE*. 803–814.
[84] R. Lapeña, J. Font, F. Pérez, and C. Cetina. 2016. Improving Feature Location by Transforming the Query from Natural Language into Requirements. In *Proc. SPLC*. 362–369.
[85] V. Lavrenko and W. B. Croft. 2001. Relevance Based Language Models. In *Proc. SIGIR*. 120–127.
[86] D. Lawrie and D. Binkley. 2018. On the Value of Bug Reports for Retrieval-Based Bug Localization. In *Proc. ICSME*. 524–528.
[87] B. Lemaire and G. Denhière. 2008. Effects of High-Order Co-occurrences on Word Semantic Similarities. *CoRR* (2008).
[88] O. A. L. Lemos, S. Bajracharya, J. Ossher, P. C. Masiero, and C. Lopes. 2011. A test-driven approach to code search and its application to the reuse of auxiliary functionality. *IST* 53, 4 (2011), 294 – 306.
[89] O. A. L. Lemos, A. C. de Paula, H. Sajnani, and C. V. Lopes. 2015. Can the use of types and query expansion help improve large-scale code search?. In *Proc. SCAM*. 41–50.
[90] O. A. L. Lemos, A. C. de Paula, F. C. Zanichelli, and C. V. Lopes. 2014. Thesaurus-based Automatic Query Expansion for Interface-driven Code Search. In *Proc. MSR*. 212–221.
[91] Z. Li, T. Wang, Y. Zhang, Y. Zhan, and G. Yin. 2016. Query Reformulation by Leveraging Crowd Wisdom for Scenario-based Software Search. In *Proc. Internetware*. 36–44.
[92] Z. Li, G. Yin, T. Wang, Y. Zhang, Y. Yu, and H. Wang. 2018. Correlation-based software search by leveraging software term database. *Frontiers of Computer Science* 12, 5 (2018), 923–938.
[93] Z. Lin, Y. Zou, J. Zhao, and B. Xie. 2017. Improving software text retrieval using conceptual knowledge in source code. In Proc. ASE. 123–134.

[94] E. Linstead, S. Bajracharya, T. Ngo, P. Rigor, C. Lopes, and P. Baldi. 2009. Sourcerer: mining and searching internet-scale software repositories. Data Mining and Knowledge Discovery 18, 2 (2009), 300–336.

[95] C. Liu, X. Xia, D. Lo, C. Gao, X. Yang, and J. C. Grundy. 2020. Opportunities and Challenges in Code Search Tools. CoRR abs/2011.02297 (2020).

[96] D Liu, A Marcus, D Poshyvanyk, and V Rajlich. 2007. Feature Location via Information Retrieval Based Filtering of a Single Scenario Execution Trace. In Proc. ASE. 234–243.

[97] J. Liu, S. Kim, V. Murali, S. Chaudhuri, and S. Chandra. 2019. Neural Query Expansion for Code Search. In Proc. MAPL. 29–37.

[98] J. Lu, Y. Wei, X. Sun, B. Li, W. Wen, and C. Zhou. 2018. Interactive Query Reformulation for Source–Code Search With Word Relations. IEEE Access 6 (2018).

[99] Mei Li, X. Sun, S. Wang, D. Lo, and Yucong Duan. 2015. Query expansion via WordNet for effective code search. In Proc. SANER. 545–549.

[100] X Allan Lu and Robert B Keefer. 1995. Query Expansion / Reduction and its Impact on Retrieval Effectiveness. In Proc. TREC. 1–9.

[101] A. D. Lucia, R. Oliveto, and P. Sgueglia. 2006. Incremental Approach and User Feedbacks: a Silver Bullet for Traceability Recovery. In Proc. ICSM. 299–309.

[102] F. Lv, H. Zhang, J. Lou, S. Wang, D. Zhang, and J. Zhao. 2015. CodeHow: Effective Code Search Based on API Understanding and Extended Boolean Model. In Proc. ASE. 260–270.

[103] A. Mahmoud and G. Bradshaw. 2015. Estimating Semantic Relatedness in Source Code. TOSEM 25, 1 (Dec. 2015), 10:1–10:35.

[104] D. Mandelin, L. Xu, R. Bodik, and D. Kimelman. 2005. Jungloid Mining: Helping to Navigate the API Jungle. In Proc. PLDI. 48–61.

[105] C. D. Manning, P. Raghavan, and H. Schütze. 2008. Introduction to Information Retrieval. Cambridge University Press, New York, NY, USA.

[106] A Marcus, A Sergeyev, V Rajlich, and J I Maletic. 2004. An Information Retrieval Approach to Concept Location in Source Code. In Proc. WCRE. 214–223.

[107] L. Martie, T. D. LaToza, and A. v. d. Hoek. 2015. CodeExchange: Supporting Reformulation of Internet-Scale Code Queries in Context (T). In Proc. ASE. 24–35.

[108] P. McCullagh and J. Nelder. 1989. Generalized Linear Models. Chapman and Hall/CRC.

[109] C McMillan, M Grechanik, D Poshyvanyk, Q Xie, and C Fu. 2011. Portfolio: Finding Relevant Functions and their Usage. In Proc. ICSE. 111–120.

[110] J. Miao, J. X. Huang, and Z. Ye. 2012. Proximity-based Rocchio’s Model for Pseudo Relevance. In Proc. SIGIR. 535–544.

[111] R Mihalcea and P Tarau. 2004. TextRank: Bringing Order into Texts. In Proc. EMNLP. 404–411.

[112] T. Mikolov, K. Chen, G. Corrado, and J. Dean. 2013. Efficient Estimation of Word Representations in Vector Space. CoRR abs/1301.3781 (2013).

[113] George A. Miller. 1995. WordNet: A Lexical Database for English. Commun. ACM 38, 11 (1995), 39–41.

[114] C. Mills, G. Bavota, S. Haiduc, R. Oliveto, A. Marcus, and A. D. Lucia. 2017. Predicting Query Quality for Applications of Text Retrieval to Software Engineering Tasks. TOSEM 26, 1 (2017), 3:1–3:45.

[115] C. Mills, J. Pantiuchina, E. Parra, G. Bavota, and S. Haiduc. 2018. Are Bug Reports Enough for Text Retrieval-Based Bug Localization?. In Proc. ICSME. 381–392.

[116] L Moreno, G Bavota, S Haiduc, M Di Penta, R Oliveto, B Russo, and A Marcus. 2015. Query-based Configuration of Text Retrieval Solutions for Software Engineering Tasks. In Proc.
A Systematic Review of Automated Query Reformulations in Source Code Search

[117] L. Moreno, J. J. Treadway, A. Marcus, and W. Shen. 2014. On the Use of Stack Traces to Improve Text Retrieval-Based Bug Localization. In Proc. ICSME. 151–160.

[118] L. Nie, H. Jiang, Z. Ren, Z. Sun, and X. Li. 2016. Query Expansion Based on Crowd Knowledge for Code Search. TSC 9, 5 (2016), 771–783.

[119] O. Panchenko, J. Karstens, H. Plattner, and A. Zeier. 2011. Precise and Scalable Querying of Syntactical Source Code Patterns Using Sample Code Snippets and a Database. In Proc. ICPC. 41–50.

[120] C. Parnin and A. Orso. 2011. Are Automated Debugging Techniques Actually Helping Programmers?. In Proc. ISSTA. 199–209.

[121] F. Perez, J. Font, L. Arcega, and C. Cetina. 2018. Automatic query reformulations for feature location in a model-based family of software products. Data & Knowledge Engineering 116 (2018), 159 – 176.

[122] F. Pérez, J. Font, L. Arcega, and C. Cetina. 2019. Collaborative Feature Location in Models through Automatic Query Expansion. AUSE 26, 1 (2019), 161–202.

[123] M. Petticrew and H. Roberts. 2005. PICO. Wiley.

[124] F. Pérez, A. C. Marcén, R. Lapeña, and C. Cetina. 2020. Evaluating Low-Cost in Internal Crowdsourcing for Software Engineering: The Case of Feature Location in an Industrial Environment. IEEE Access 8 (2020), 65745–65757.

[125] F. Pérez, T. Ziadi, and C. Cetina. 2020. Utilizing Automatic Query Reformulations as Genetic Operations to Improve Feature Location in Software Models. TSE (2020).

[126] D. Qiu, B. Li, S. Ji, and H. Leung. 2014. Regression Testing of Web Service: A Systematic Mapping Study. ACM Comput. Surv. 47, 2 (2014), 21:1–21:46.

[127] M. Raghothaman, Y. Wei, and Y. Hamadi. 2016. SWIM: Synthesizing What I Mean: Code Search and Idiomatic Snippet Synthesis. In Proc. ICSE. 357–367.

[128] A. Rahman, R. Mahdavi-Hezaveh, and L. Williams. 2018. Where Are The Gaps? A Systematic Mapping Study of Infrastructure as Code Research. (2018).

[129] M. M. Rahman, F. Khomh, and M. Castelluccio. 2020. Why are Some Bugs Non-Reproducible? An Empirical Investigation using Data Fusion. In Proc. ICSME. 12.

[130] M. M. Rahman, F. Khomh, S. Yeasmin, and C. K. Roy. 2021. The Forgotten Role of Search Queries in IR-based Bug Localization: An Empirical Study. EMSE (2021).

[131] M. M. Rahman, F. Khomh, S. Yeasmin, and C. K. Roy. 2021. The Forgotten Role of Search Queries in IR-based Bug Localization: An Empirical Study. EMSE (2021), 57.

[132] M M Rahman and C K Roy. 2014. On the Use of Context in Recommending Exception Handling Code Examples. In Proc. SCAM. 285–294.

[133] M M Rahman and C K Roy. 2015. TextRank Based Search Term Identification for Software Change Tasks. In Proc. SANER. 540–544.

[134] M M Rahman and C K Roy. 2016. QUICKAR: Automatic Query Reformulation for Concept Location Using Crowdsourced Knowledge. In Proc. ASE. 220–225.

[135] M M Rahman and C K Roy. 2017. Improved Query Reformulation for Concept Location using CodeRank and Document Structures. In Proc. ASE. 428–439.

[136] M M Rahman and C K Roy. 2017. STRICT: Information Retrieval Based Search Term Identification for Concept Location. In Proc. SANER. 79–90.

[137] M. M. Rahman and C. K. Roy. 2018. Effective Reformulation of Query for Code Search using Crowdsourced Knowledge and Extra-Large Data Analytics. In Proc. ICSME. 516–527.

[138] M. M. Rahman and C. K. Roy. 2018. Improving IR-Based Bug Localization with Context-Aware Query Reformulation. In Proc. ESEC/FSE. 621–632.
[139] M M Rahman, C K Roy, and D Lo. 2016. RACK: Automatic API Recommendation using Crowdsourced Knowledge. In Proc. SANER. 349–359.
[140] S. E. Robertson. 1991. On Term Selection for Query Expansion. J. Doc. 46, 4 (1991), 359–364.
[141] J J Rocchio. [n. d.]. The SMART Retrieval System—Experiments in Automatic Document Processing. Prentice-Hall, Inc. 313–323 pages.
[142] M Roldan-Vega, G Mallet, E Hill, and J A Fails. 2013. CONQUER: A Tool for NL-Based Query Refinement and Contextualizing Code Search Results. In Proc. ICSM. 512–515.
[143] Julia Rubin and Marsha Chechik. 2013. A Survey of Feature Location Techniques. In Domain Engineering, Product Lines, Languages, and Conceptual Models.
[144] C. Sadowski, K. T. Stolee, and S. Elbaum. 2015. How Developers Search for Code: A Case Study. In Proc. ESEC/FSE. 191–201.
[145] R. K. Saha, J. Lawall, S. Khurshid, and D. E. Perry. 2014. On the Effectiveness of Information Retrieval Based Bug Localization for C Programs. In Proc. ICSME. 161–170.
[146] R K Saha, M Lease, S Khurshid, and D E Perry. 2013. Improving Bug Localization using Structured Information Retrieval. In Proc. ASE. 345–355.
[147] G. Salton and C. Buckley. 1997. Readings in Information Retrieval. Chapter Improving Retrieval Performance by Relevance Feedback, 355–364.
[148] G. Salton and M. J. McGill. 1986. Introduction to Modern Information Retrieval.
[149] G Salton, A Wong, and C S Yang. 1975. A Vector Space Model for Automatic Indexing. Commun. ACM 18, 11 (1975), 613–620.
[150] A. Satter and K. Sakib. 2016. A search log mining based query expansion technique to improve effectiveness in code search. In Proc. ICIT. 586–591.
[151] T Savage, M Revelle, and D Poshyvanyk. 2010. FLAT3: Feature Location and Textual Tracing Tool. In Proc. ICSE. 255–258.
[152] H. A. Shafiq and Z. Arshad. 2014. Automated Debugging and Bug Fixing Solutions : A Systematic Literature Review and Classification.
[153] D Shepherd, Z P Fry, E Hill, L Pollock, and K Vijay-Shanker. 2007. Using Natural Language Program Analysis to Locate and Understand Action-Oriented Concerns. In Proc. ASOD. 212–224.
[154] Z Shi, J Keung, and Q Song. 2014. An Empirical Study of BM25 and BM25F Based Feature Location Techniques. In Proc. InnoSWDev. 106–114.
[155] A. Shtok, O. Kurland, D. Carmel, F. Raiber, and G. Markovits. 2012. Predicting Query Performance by Query-Drift Estimation. TOIS 30, 2 (2012), 11:1–11:35.
[156] R. Sirres, T. F. Bissyandé, D. Kim, D. Lo, J. Klein, K. Kim, and Y. L. Traon. 2018. Augmenting and structuring user queries to support efficient free-form code search. EMSE (2018), 2622–2654.
[157] Bunyamin Sisman, Shayan A. Akbar, and Avinash C. Kak. 2017. Exploiting spatial code proximity and order for improved source code retrieval for bug localization. JSEP 29, 1 (2017), e1805.
[158] B Sisman and A C Kak. 2012. Incorporating Version Histories in Information Retrieval Based Bug Localization. In Proc. MSR. 50–59.
[159] B Sisman and A C Kak. 2013. Assisting Code Search with Automatic Query Reformulation for Bug Localization. In Proc. MSR. 309–318.
[160] G Sridhara, E Hill, L Pollock, and K Vijay-Shanker. 2008. Identifying Word Relations in Software: A Comparative Study of Semantic Similarity Tools. In Proc. ICPC. 123–132.
[161] K. Stol, P. Ralph, and B. Fitzgerald. 2016. Grounded Theory in Software Engineering Research: A Critical Review and Guidelines. In Proc. ICSE. 120–131.
[162] K. T. Stolee, S. Elbaum, and D. Dobos. 2014. Solving the Search for Source Code. TOSEM 23, 3 (2014), 26:1–26:45.
[163] F Thung, D Lo, and J Lawall. 2013. Automated Library Recommendation. In Proc. WCRE. 182–191.
[164] F Thung, S Wang, D Lo, and J Lawall. 2013. Automatic Recommendation of API Methods from Feature Requests. In Proc. ASE. 290–300.
[165] V. Vinayakarao, A. Sarma, R. Purandare, S. Jain, and S. Jain. 2017. ANNE: Improving Source Code Search Using Entity Retrieval Approach. In Proc. WSDM. 211–220.
[166] J. Wang, X. Peng, Z. Xing, and W. Zhao. 2013. Improving Feature Location Practice with Multi-faceted Interactive Exploration. In Proc. ICSE. 762–771.
[167] Q Wang, C Parnin, and A Orso. 2015. Evaluating the Usefulness of IR-based Fault Localization Techniques. In Proc. ISSTA. 1–11.
[168] S Wang and D Lo. 2014. Version History, Similar Report, and Structure: Putting Them Together for Improved Bug Localization. In Proc. ICPC. 53–63.
[169] S. Wang and D. Lo. 2016. AmaLgam+: Composing Rich Information Sources for Accurate Bug Localization. JSEP 28, 10 (2016), 921–942.
[170] S. Wang, D. Lo, and L. Jiang. 2014. Active Code Search: Incorporating User Feedback to Improve Code Search Relevance. In Proc. ASE. 677–682.
[171] S. Wang, D. Lo, and L. Jiang. 2016. AutoQuery: automatic construction of dependency queries for code search. ASE 23, 3 (Sep 2016), 393–425.
[172] T. Wei, Y. Lu, H. Chang, Q. Zhou, and X. Bao. 2015. A semantic approach for text clustering using WordNet and lexical chains. Expert Systems with Applications 42, 4 (2015), 2264 – 2275.
[173] M Wen, R Wu, and S C Cheung. 2016. Locus: Locating bugs from software changes. In Proc. ASE. 262–273.
[174] L A Wilson. 2010. Using Ontology Fragments in Concept Location. In Proc. ICSM. 1–2.
[175] C P Wong, Y Xiong, H Zhang, D Hao, L Zhang, and H Mei. 2014. Boosting Bug-Report-Oriented Fault Localization with Segmentation and Stack-Trace Analysis. In Proc. ICSME. 181–190.
[176] W. E. Wong, R. Gao, Y. Li, R. Abreu, and F. Wotawa. 2016. A Survey on Software Fault Localization. TSE 42, 8 (2016), 707–740.
[177] H. Wu and Y. Yang. 2019. Code Search Based on Alteration Intent. IEEE Access 7 (2019), 56796–56802.
[178] M. Wursch, G. Ghezzi, G. Reif, and H. C. Gall. 2010. Supporting developers with natural language queries. In Proc. ICSE. 165–174.
[179] J Yang and L Tan. 2012. Inferring Semantically Related Words from Software Context. In Proc. MSR. 161–170.
[180] J. Yang and L. Tan. 2014. SWoRNet: Inferring semantically related words from software context. EMSE 19, 6 (2014), 1856–1886.
[181] X Ye, H Shen, X Ma, R Bunescu, and C Liu. 2016. From Word Embeddings to Document Similarities for Improved Information Retrieval in Software Engineering. In Proc. ICSE. 404–415.
[182] K C Youm, J Ahn, J Kim, and E Lee. 2015. Bug Localization Based on Code Change Histories and Bug Reports. In Proc. APSEC. 190–197.
[183] H. Yu, W. Song, and T. Mine. 2016. APIBook: An Effective Approach for Finding APIs. In Proc. Internetware. 45–53.
[184] T Yuan, D Lo, and J Lawall. 2014. Automated Construction of a Software-Specific Word Similarity Database. In Proc. CSMR-WCRE. 44–53.
[185] S. Zamani, S. Peck Lee, R. Shokripour, and J. Anvik. 2014. A noun-based approach to feature location using time-aware term-weighting. Information and Software Technology 56, 8 (2014), 991 – 1011.
[186] F. Zhang, H. Niu, I. Keivanloo, and Y. Zou. 2018. Expanding Queries for Code Search Using Semantically Related API Class-names. *TSE* 44, 11 (2018), 1070–1082.

[187] Jie Zhang, XiaoYin Wang, Dan Hao, Bing Xie, Lu Zhang, and Hong Mei. 2015. A survey on bug-report analysis. *SCIS* 58, 2 (2015), 1–24.

[188] W. Zhang, Z. Li, Q. Wang, and J. Li. 2019. FineLocator: A novel approach to method-level fine-grained bug localization by query expansion. *IST* 110 (2019), 121–135.

[189] Y. Zhang, D. Lo, X. Xia, G. Scanniello, T. B. Le, and J. Sun. 2018. Fusing multi-abstraction vector space models for concern localization. *EMSE* 23, 4 (2018), 2279–2322.

[190] J Zhou, H Zhang, and D Lo. 2012. Where Should the Bugs Be Fixed? - More Accurate Information Retrieval-based Bug Localization Based on Bug Reports. In *Proc. ICSE*.

[191] W. Zou, D. Lo, Z. Chen, X. Xia, Y. Feng, and B. Xu. 2018. How Practitioners Perceive Automated Bug Report Management Techniques. *TSE* (2018), to appear.