

The University of Saskatchewan
Department of Computer Science

Technical Report #2014-02



UNIVERSITY OF
SASKATCHEWAN

Towards Source Code Clone Search via Information Retrieval

Iman Keivanloo, Chanchal K. Roy, and Juergen Rilling

Abstract—Finding both pattern and content similarities in source code constitutes the core of several research domains such as plagiarism and clone detection. Recently, code clone search as a new research branch has emerged aiming to provide similarity search functionality for code fragments. Scalability, short response time, and the ability to search for Type-1, 2 and 3 clones are some of the major challenges which have to be dealt with by the clone search community. We introduce a clone search model, based on clone detection and information retrieval that not only provides scalability, short response times, and Type-1, 2 and 3 detection, but also supports result ranking. Ranking of result sets is a key functionality, which has been widely overlooked previously by the clone search community. For our approach, the ranking not only forms the basis for ordering result sets based on their similarity to a given query but also introduces an important quality concept in clone search beyond traditional precision and recall measures. Our model takes advantage of a non-positional multi-level indexing approach to achieve a search that is scalable with a high recall. Result sets are ranked using two information retrieval ranking approaches: Jaccard similarity coefficient and cosine similarity. Both ranking approaches exploit code patterns' and not tokens to derive local and global frequencies. We studied the performance of our search engine through 40 candidate search schemata for Type-1, 2, and 3 clones on a dataset covering 25,000 projects. Through benchmarking and various measures, we observed that clone search via information retrieval is capable to deliver scalability with high precision and recall in near real-time.

Index Terms— Source code clone search, source code similarity, information retrieval, clone detection, source code search



1 INTRODUCTION

The term clone (Greek word klōn) dates back to Herbert J. Webber's [WEH03] work in 1903, referring to the outcome of a derivation activity in the context of living species. While in computer science, such autonomous reproduction is limited, derivation is an unavoidable fact of programming and is known as cloning. Derivation during software development usually occurs as the result of reuse [PER88] [DEE05]. The ease of reuse and the potential harms caused by cloning in software development became a major motivation for computer scientists to investigate this type of code duplications. Consequently, a research discipline - clone detection - [BEL07] [ROS09] [BAK92] has emerged in computer science, which focuses on devising novel algorithms and heuristics for finding, tracing, and managing [KOS08] code clones. Although the input data for this type of similarity search is source code, which is structured and well organized, the clone detection problem remains a non-trivial problem due to the different types of similarities that can be distinguished [BEL07]. At source code level, clones share two types of similarity: (1) pattern and (2) content similarity. The challenge lies often in determining if two code fragments (e.g., "int temp=0;" and "float f=2) are actually cloned, as they can hold only negligible content (e.g., token names) similarity.

More recently, *clone search* (e.g., [LER10]) has emerged as a new research direction that exploits the fundamentals of clone detection research to provide (similarity) search

functionality for code fragments (i.e., clones). In contrast to traditional clone detection, clone search is only concerned with locating similar code fragments for a given code fragment at run-time. In the literature, several terms have been introduced to emphasize the importance of response time in clone search, e.g., just-in-time [BAR10], real-time [KAW09], and instant [LER10] clone search. Several similarity and search models (exploiting clone detection fundamentals) have been proposed to address the core requirements of clone search: scalability, short response time, and being able to search for Type-1, 2 and 3 clones. Similar to other search domains (e.g., Web search), clone search models are dealing with large search spaces returning often hundreds of matches for each query [KLX11]. We therefore believe that ranking of results sets in clone search, as it is already in other domains, has to be considered a core requirement.

In this research, we introduce a clone search model that covers ranking, scalability, fast response time, and Type-1, 2, and 3 detection. This model is based on our earlier work on clone search [KLX11] [KLZ11] and its emerging applications such as code search (e.g., [KLX12]). Our studies in [KLX11] have shown that a multi-level indexing approach can achieve scalability, short response time, and search capabilities for Type-1, 2 and 3 clones. However, support for ranking was missing.

For this research, we extended our multi-level indexing approach by adopting the Jaccard similarity coefficient [JAC01] and cosine similarity [MAN08] to provide ranking of the result sets. Thus clone search ranking approach exploits code patterns' (not token) local and global frequencies for assigning different weights to search re-

• Iman Keivanloo, Queen's University. E-mail: iman.keivanloo@queensu.ca.
 • Chanchal K. Roy, University of Saskatchewan. E-mail: croy@cs.usask.ca.
 • Juergen Rilling, Concordia University. E-mail: rilling@concordia.ca.

sults, depending on the popularity of the patterns. For example, a domain specific pattern (e.g., “EclipseEditor foo=new EclipseEditor()”) can be assigned higher weights compared to some general code patterns (e.g., “catch (Exception ex) {”). Moreover, the adaptation of Jaccard and cosine similarity makes it possible to discard all positional information during indexing and pattern matching. Discarding such positional information improves the scalability and performance in terms of memory consumption for indexing and computational complexity of the pattern matching.

We have studied the applicability of our similarity search model using a representative dataset of 25,000 open source Java projects. We evaluated the performance (scalability, response time, ability to detect Type-1, 2, and 3 clones as well as the ranking of result sets) of our search model, using 40 different clone search configurations.

For the evaluation of the ranking, we compared these schemata based on their ability to provide result sets in which the correct and most relevant search results were consistently placed in the top of these result sets. The evaluation was performed on a large corpus containing approximately 356 Million LOC, and a clone benchmark, which includes 50 queries and 650 seeded Type-1, 2, and 3 clones. Our evaluation study also includes an extensive manual relevance scoring for 117,000 results, which were extracted from the top-60 hits of over 2000 experimental queries. As part of this manual evaluation, we analyzed the relevancy of 80,000 hits using a predefined scoring guideline, comparing both their clone types and their similarity with the search query.

We selected 5 measures from the IR literature, to evaluate different quality aspects of our clone search model and to identify outperforming schemata. We observed that our approach is scalable; there are certain configurations of our model that are able to answer a given query in real-time (~100 milliseconds). The identified configurations also (1) detect all of the known Type-1, 2 and 3 clones (2) and successfully rank them at top of the result set.

The remainder of the paper is organized as follows. Section 2 outlines related work in clone detection, similarity search and code search. Section 3 introduces our clone SeClone search model. Section 4 discusses retrieval and indexing steps of our search model in details, with the different ranking schemata of our search model being covered in Section 5. Section 6, provides a discussion on the data characteristics of the corpus being search in our domain of discourse. Section 7 introduces the measures we adopted from other domains to support our qualitative analysis of clone search results (e.g., information retrieval). Finally Sections 8, 9, 10 and 11 provide the benchmark preparation, performance evaluation results, followed by threats to validity and conclusions.

2 BACKGROUND AND RELATED WORK



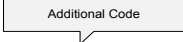
In this section, we present a review of both early research and the state of the art in (1) source code clone, (2) code clone detection, and (3) code similarity.

2.1. Background

One of the earliest similarity detection approaches dates back to the work by Ottenstein [OTT76] in 1976. Ottenstein introduced a metric-based approach for the detection of plagiarism in student programming assignments. His work also included a discussion on potential dissimilarity types that were supported by a plagiarism detection algorithm, such as re-formatting, re-naming and re-ordering of statements. Later on, Grier [GRI81] in 1981 extended Ottenstein’s work to Pascal code. The first actual reference to the clone concept in the source code and programming domain dates back to the work by Abrams and Myrna [ABR79] in 1979. They used the term clone in a Programming Language (APL) context describing it as “... creates an output file and starts a "clone" of itself”. The concept of a “clone” in source code was later used by Jacobsen [JAC84] to describe a pre-defined command, and by Caudill and Wirfs-Brock [CAU86] as a reproduction of executable files in Smalltalk. Tanenbaum [TAN87] used clone to describe the variations of a software system. During the 1980s, the term clone was further popularized mostly through its use as a reference to computer hardware, such as compatible computer (hardware), an IBM compatible (or short IBMclone) computer [KEL83] or, in [LOM83], as “...can’t tell what is on my disk without a clone of my computer”. Among the first researchers who actually used the term clone detection at the source code level were Carter et al. in 1993 [CAR93]. They described clone detection as the process of finding similar telecommunications systems using neural networks.

Over the last two decades, clone detection, the process of finding code duplications in programming content [ROS09], has matured as a research discipline in computer science and resulted in a number of clone detection techniques. Common to these traditional detection techniques is that they perform a complete off-line search step to find all possible clone pairs within a static source code repository. At source code level, clones share two types of similarity: (1) pattern and (2) content. Table 1 provides an overview, including examples, of the three basic similarity types related to syntactical clones.

Table 1. Examples for source code similarity types (i.e., clone types)

The input code sample	
HashMap var=new HashMap (10);	
Similarity Type	Example
Type-1	<div style="text-align: center;">  </div> <pre>HashMap var = new HashMap (10);</pre>
Type-2	<div style="text-align: center;">  </div> <pre>HashMap list1=new HashMap ();</pre>
Type-3	<div style="text-align: center;">  </div> <pre>HashMap list1=new HashMap (list2.size());</pre>

These are clones (Table 1) with an observable similarity in the source code. Type-1 clones are exact copies of each other, except for possible differences in whitespaces

and comments. Type-2 clones are parameterized copies, where variable names and function calls have been renamed and/or types have been changed. Changes (e.g., addition and deletion of statements) in a clone pair result in type-3 clones. In cases where two fragments share similar functionality with different syntactical presentations, they constitute a Type-4 clone pair.

2.2. Related work

While earlier work in code clone and similarity research had mainly focused on detecting plagiarism in source code, this focus started to shift in the 1990s with software maintenance emerging as a new application for clone detection. In 1992, Baker [BAK92] proposed Dup, a tool to support software maintenance and bug fixing by detecting duplicate code. The Dup tool also implemented a clone detection solution, which exploited hash values and inverted-indexes to facilitate the search process during clone detection. Later approaches, such as metric-based in Merlo et al. in 1996 [MAY96] and AST-based Baxter et al. in 1998 [BAX98], supported the use additional facts extracted from source to further improve scalability, performance, and efficiency of clone detection approaches.

Clone Detection. Most clone detection approaches (e.g., CCFinder [KAM02]) are based on sequence comparison. Data dependency and program dependency graph (PDG) are also studied for clone detection as alternative representations of computer program (e.g., Jia et al. [JMM09] or Higo and Kusumoto [YHU11]). Recently, novel search and retrieval models were introduced to scale clone detection to larger corpora such as scalable clone detection using suffix trees by Koschke [KOS12] and Göde [GRK09], R* tree by Jiang et al. [JIA07], simhash by Uddin et al. [UDD11][UDD13] and Levenshtein metric and Manhattan distance by Lavoie and Merlo [LAV11][LAV12]. Similarity measures and ranking for clone detection is also studied by Smith and Horwitz [SMI09]. There is also a body of work on the other forms of clones such as structural clones by Abdul Basit and Jarzabek [ABJ10] and semantic clones by Kim et al. [KKI11] and Gabel, Jiang, and Su [GJZ08]. Several applications for clone detection are discussed in the literature such as maintenance improvement by pro-active clone detection by Jablonski and Hou [JAH10].

While the existing research is focused on clone detection, it also forms the foundations for any research about code similarity measurement and clone search. Our research approach is similar to the one by Carter et al. [CAR93] since both use cosine similarity. Our approach also shares commonalities with the vector space models used by DECKARD [JIA07] and Carter et al. [CAR93]. However, we create the vectors using code patterns instead of metrics and predefined code fingerprints [JIA07] [CAR93]. Also compared to the work in NiCad [ROS08] and CCFinder [KAM02]), we deploy a non-positional similarity search instead of a sequence matching approach. This approach provides us the possible of achieving scalability and real-time response time both together.

Non-positional retrieval is explored earlier by Smith and Horwitz [SMI09], Baker et al. [BAK98], and Uddin et al. [UDD11][UDD13]. Our research focuses on the application of Information Retrieval models for clone search. Our approach not only detects the major clone types but also is aiming to discriminate among Type-1, 2 and 3 clones. Being able to differentiate between Type-1, 2, and 3 is important if we want to rank the detected fragments based on their similarity to the query.

In other research domains such as concept location, information retrieval has been explored for code similarity. Marcus and Maletic [MAR01] used Latent Semantic Indexing (LSI) to extract semantics from source code facts (e.g., identifier names) to guide the detection of code fragments implementing similar features. LSI also has been exploited by Tairas and Gray [TAI09] for clone result clustering. Kontogiannis [KON97] uses a basic IR infrastructure and Mishne et al. [MIS04] introduced an approach using Conceptual Graphs and structural information to find similar code.

Clone Search. Although detecting code similarities and patterns is a well-established research area in clone detection, more recently “source code clone search”, a research area also known as just-in-time [BAR10], real-time [KAW09], or instant [LER10] clone search has emerged. While clone search still shares its fundamentals with traditional clone detection, both its objective and requirements differ significantly. Traditional clone detection applications are based on a complete off-line search step to find all possible clone pairs within a static source code repository. In contrast, code clone search models can be considered as specialized search engines that are designed to find clones matching a single fragment (query) within an often large corpus. Clone search approaches index source code repositories as part of their off-line processing and use input provided in the form of a code fragment at run-time, to trigger and perform the search process.

SHINOBI [KAW09] provides the search functionality via a suffix array built on transformed tokens using CCFinder’s rules [KAM02]. Hummel et al. [HUM10] use inverted index for scalable Type-2 clone search. A multi-dimensional token-level indexing approach is introduced by Lee et al. [LER10] [LEM11] using an R* tree on DECKARD’s [JIA07] approximate vector matching. The language elements (e.g., assignment) constitute the dimensions of the search space. Barbour et al. [BAR10] introduce a result sampling approach that uses results obtained from other clone detection tools to find candidate clones. The collected candidates are indexed and then compared by Knuth-Morris-Pratt string searching algorithm [KNU77]. Zibran and Roy [ZIB12] introduced an IDE-support for Type-3 clone search based on Rabin’s fingerprinting algorithm and suffix trees. Bazrafshan and Koschke [BAZ11] exploit Chang and Lawler’s search algorithm, which was originally proposed for the bioinformatics domain to find approximate source code patterns.

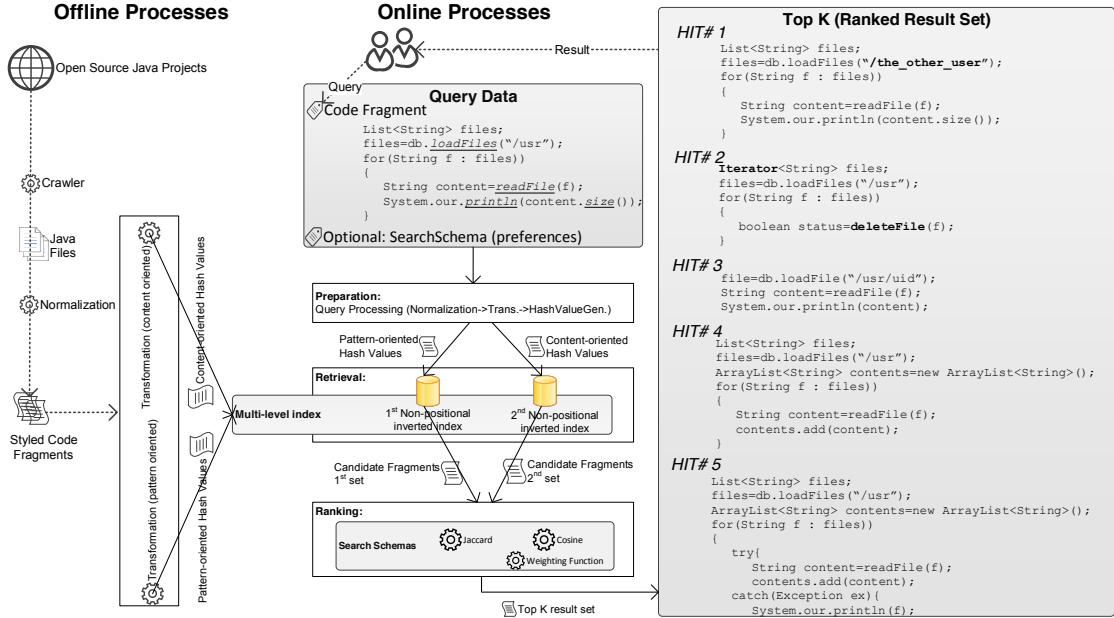


Figure 1. SeClone –clone search approach

In our earlier work on clone search [KLX11] [KLZ11], we introduced a hash-based inverted indexing approach, which uses multi-level indexing for Type-3 clone search. Our research differs from earlier work (including ours) on clone search since we are aiming to provide a clone search model that includes not only support for Type-2 and 3 clones but also ranking of the result set. Moreover, we are proposing adaptation of non-positional search space of code patterns and information retrieval models for both retrieval and ranking. To the best of our knowledge, Jaccard coefficient and vector space model (combined with local and global frequencies) via cosine similarity for a set of vectors made of code patterns have not been studied before for clone search.

3 A MODEL FOR CLONE SEARCH

In this section, we introduce our code clone search approach, SeClone, which supports up to Type-3 clone, scalability, fast response time, and ranking of result sets. Our model is based on existing models, including vector space model (VSM), cosine similarity, and Jaccard similarity coefficient (JSC) from information retrieval (IR). They are frequently used by the IR community for similarity search due to their scalability to large corpora [BRI98] [MAN08].

We are motivated to adapt them in our research since common to both models is low computational complexity for Type-3 clone search process. The low complexity is due to their non-positional approach. The non-positional similarity search is different from the dominant matching approach in clone detection that is positional (e.g., longest common subsequent (LCS) [HUN77] and suffix tree).

3.1 Overview

SeClone combines multi-level indexing and information retrieval ranking models. Our approach is able to find the closest matches (hits) to a given query, and return these hits as a ranked result set based on their similarity to the

input query. Figure 1 provides an overview of SeClone and its major processing steps, which include: (1) preprocessing, (2) indexing, (3) retrieval and (4) ranking. The performance (off-line and online processing) of our search model approach is configurable via its search schema, which consists of nine parameters (Figure 2).

Preprocessing. SeClone is a line based detection approach that uses abstract syntax tree (AST) as its input for the offline preprocessing step. SeClone parses the ASTs of individual files to create a uniform representation, annotated by token types. The preprocessing step also transforms AST tokens using transformation rules, which are specified through the search schema parameters t_p and t_s . These transformation rules generate an encoded code patterns (ep) for each line of code.

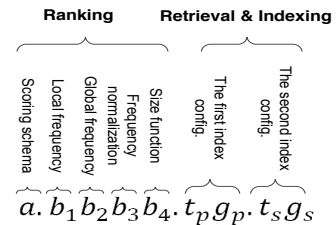


Figure 2. The SeClone search schema - configuration parameters

Indexing. Our approach uses multi-level indexing. By default, it creates two¹ inverted indices, denoted by p (primary) and s (secondary). The ep datasets, generated by the transformation rules t_p and t_s , are indexed as hash table-based indices. The hash values can be generated for different granularities: g_p and g_s which are specified as part of the search schema.

Retrieval. At run-time, SeClone creates two ep datasets for the given query (i.e., code fragment). The hash values

¹ Our multi-level indexing idea implies that the actual number of indices should be at least two when both pattern and content similarity are important (e.g., Type-3 clone search). However, depending on the actual application context, additional indices can be added.

are generated using the same configuration used by the preprocessing and indexing steps. Finally, SeClone generates two vectors ($v_{primary}$ and $v_{secondary}$) for each query q using hash values of the encoded code patterns:

$$\begin{aligned} \text{Input (ordedred bag):} \quad & q (l_1, \dots, l_y) \\ & v_{primary} < ep_1^p, ep_2^p, ep_3^p, \dots, ep_n^p > \\ & v_{secondary} < ep_1^s, ep_2^s, ep_3^s, \dots, ep_n^s > \end{aligned}$$

These vectors ignore the ordering of the elements similar to our inverted indices. For each vector, a look up action is performed on the corresponding index. The goal is to retrieve all code fragments indexed in the corpus, which share at least one hash value ep_x^y with the query. The union of both candidate sets, from the primary and secondary indices, constitutes the complete set of hits, i.e., all clone candidates.

Ranking. The retrieval step finds all possible answers for a given query. However, all of them are not equally similar to the query. In addition, in a comprehensive corpus most of the candidates are false positives. The goal of ranking step is to sort them based on their similarity degree to the query (e.g., Figure 1). Without a proper ranking, the end user has to iterate over thousands of matches to find the true positives. Our ranking models are based on VSM and JSC, which both can be configured within our search schema ($\mathbf{a}, \mathbf{b}_1 \mathbf{b}_2 \mathbf{b}_3 \mathbf{b}_4, t_p g_p, t_s g_s$), with the ranking parameters being highlighted in bold. Similarity scores are calculated for each hit that is returned by the retrieval step. SeClone generates the final ranked result set (e.g., Figure 1) by sorting the hits over the calculated similarity degree. Figure 3 summarizes the SeClone search algorithm for both retrieval and ranking steps.

Algorithm *Retrieval_and_Ranking*($q, ix^p, ix^s, a, b_1 b_2 b_3 b_4, t_p g_p, t_s g_s$)

Input q : query's code fragment, ix^p : primary and secondary indices

Output $hits$: ordered set of all candidate clone fragments based on their similarity to the query

1. $v_{primary}[] \leftarrow HashValue(q, t_p, g_p)$ // $v_{primary}$: the un-ordered set of hash values
2. $v_{secondary}[] \leftarrow HashValue(q, t_s, g_s)$
3. **for** h **in** $v_{primary}$ // find and add all fragments with at least one occurrence of h
4. $hits_{primary}[] \leftarrow ix^p.lookup(h)$
5. **for** h **in** $v_{secondary}$
6. $hits_{secondary}[] \leftarrow ix^s.lookup(h)$ // this is an un-ordered set of all candidate clones
7. $hits[] \leftarrow hits_{primary} \cup hits_{secondary}$
8. **for** hit **in** $hits$
9. $hits'[i] \leftarrow relevance_score(q, hit, a, b_1, b_2, b_3, b_4)$
10. $sort(hits'$ on $relevance_score)$
11. **return** $hits$

Figure 3. Algorithm overview for retrieval and ranking steps

3.2. Computational complexity

A summary of the computational complexity of our approach for both run-time complexity and memory consumption is shown in Table 2. For the analysis, we excluded style unification, transformations, and AST build times, since they are negligible and linear to the size of the input data set.

We separate our analysis in three major processing steps: (1) off-line indexing for creating the hash table indices, (2) the actual search, which includes retrieval and ranking, and (3) the corpus update. T represents the inverted index size, which is $O(n)$ with n being the size of

the corpus in terms of lines of code (LOC). The size of the result set is represented by c , and the total number of updated lines of code by l , with the expected lookup complexity for the inverted index being $O(1)$, since the index is hash table-based. The resulting clone search time complexity is $O(c * \log c)$, since in order to create the ranked result set all hits must be first sorted based on their relevance scores. This low time complexity for both clone search (including Type-3 clones) and repository preparation can be attributed to the use of non-positional indexing. Memory consumption for indices is also almost linear. This cannot be further optimized without the use of compression and other abstraction mechanisms.

Table 2. SeClone computational complexity

Processing step	Time complexity	Memory complexity
Repository preparation (Indexing)	$O(n)$	$O(n)$
Clone search	$O(c * \log c)$	$O(c)$
Repository update (content addition/deletion)	$O(l)$	$O(l + T_l)$

4. SECLONE INDEXING MODEL

In this section, we describe the details of our indexing approach. Our clone search model uses the concept of encoded code pattern (ep) to construct its search space. An encoded code pattern is a template that defines a certain degree of similarity. The idea of encoding code pattern supports Type-2 detection following Baker's *p-strings* [BAK92]. However, instead of using these patterns directly, they are transformed to hash values. Hash values provide an efficient numeric representation of textual content in terms of space consumption and retrieval (lookup) times, with a lookup complexity of $O(1)$. Both of these properties are important for our model to ensure that it is both scalable and efficient.

4.1. Encoded code pattern generation

Our encoded code patterns are based on line granularity. Encoding the original code content "as is", would constitute the most restrictive ep , and only allow to detect/match exact (Type-1) clones during the search process. Less restrictive encoding increases recall and supports Type-2 clone search however, at the cost of lower precision. In our research, we defined a number of models for encoding code patterns to address the tradeoff between recall and precision. Each model is defined through (1) a transformation function and (2) its encoding granularity. The granularity (g) determines the number of neighboring lines of code that will be considered for the encoding. The transformation function t , on the other hand, determines the parameterization rules.

Hash function. The hash function H is used for generating hash values that represent the encoded code pattern. The function uses four input parameters: the code fragment c , offset o , granularity g , and the transformation function t .

$$H(c, o, g, t) = v$$

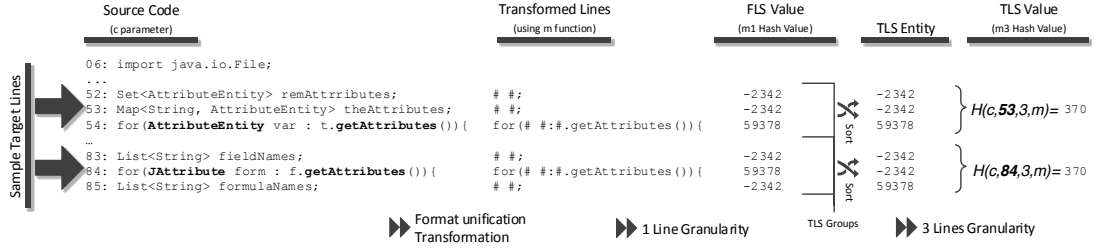


Figure 4. Example of SeClone's hash and transformation function output

Since our approach is a line-based clone search, the hash function also operates at line level granularity. Consequently, the input code fragment has to be at least one syntactically complete line of code. The offset refers to the line of code that is used as a target line for the hash value generation process. The generation of hash values for a code fragment requires the function to be called several times, by iterating over the target line parameter.

Granularity. The H function generates hash values not only based on the target line content, but also on its neighboring content. While having a single line granularity can increase recall, such fine-granularity often decreases precision, as the overall similarity depends not only on the resemblance of the participating lines but also on their order. Therefore, to improve search precision, code patterns should also be encoded for higher granularity levels. In our approach, can generate hash values for a target line at both one and three-line granularities (Table 3).

Transformation functions. Table 4 reviews the transformations (t) supported by our approach, including their semantics, i.e., type of transformation being performed. A key difference among these transformations is their emphasis on either content or pattern resemblance. While content resemblance focuses on token name similarities, pattern resemblance enforces the order of tokens regardless of the token names. For example, the transformation function w ignores the token ordering completely, while m attempts to keep the balance between patterns and content resemblance.

Example. For line-based detection approaches, code layout unification through formatting and normalization is an essential processing step to increase recall of the retrieval algorithm [KAM02]. Layout unification requires normalization of all source code extracted from the code repository and the search queries. During the layout normalization, information from the AST of each source code file in the repository is used to extract tokens and data types. The extracted information is the input to the transformation functions. Furthermore, a combination of transformation function and granularity parameters can be used to specify a specific encoding model. For example, $m3$ refers to the TLS granularity using the transformed lines of code with only method name preservation. Figure 4 illustrates the complete process how our hash function assigns an identical value to two different code fragments by exploiting the $m3$ encoded pattern model. In this case, the code fragments identified by the target lines 53 (i.e., lines 52-54) and 84 (i.e., lines 83-85) share the same pattern but their content resemblance is low due to differing class and variable names (Figure 4).

Table 3. SeClone pre-defined granularities for hash function $-g$

	Granularity	Description
FLS	1	Only the target line that is specified by the offset parameter must be considered
TLS	3	The target line specified by the offset parameter o including $o - 1$ and $o + 1$ lines must be considered - Three lines in total

Table 4. SeClone source code transformation functions – the t parameter

Transformation function	Description	Style unification	Preserve code ordering within line	Preserve method call names e.g.,	Preserve class names e.g., Stream	Preserve symbols e.g., ()	Preserve primitive types e.g., int	Preserve language keywords	Preserve constants and literals	Preserve variable names
exact										
x	Same as input except for changes in style	x	x	x	x	x	x	x	x	x
loose (Type-1)										
l	Same content for all code fragments which can be considered as Type-1 clone	x	x	x	x	x	x	x	-	x
word set										
w	An unordered set of the selected fingerprints (only method and type tokens)	x	-	x	x	-	x	-	-	-
transformed tokenized method fingerprints										
m	Preserves only method names in method call tokens and the overall pattern, while the content (i.e., names) of the other tokens are ignored via replacing them by a single place holder (e.g., #).	x	x	x	-	x	x	x	-	-
transformed tokenized method and type fingerprints										
c	Similar behavior as m except it preserves the content of both method and type tokens.	x	x	x	x	x	x	x	-	-

4.2. Non-positional multi-level indexing and retrieval

In this section, we discuss the motivation behind our idea for non-positional multi-level indexing. The encoded code patterns represented by hash values support the detection

of two categories of similarities, pattern and content similarity. Figure 4 provides an example of two cloned fragments which are identified as similar using the *m3* model. Standard hash value-based indexing and retrieval approaches can identify these two code fragments as clones. However, if a third fragment exists in the corpus that is identical to the first fragment (line 52-54), a single indexing model using a single encoded code pattern will not be capable of distinguishing potential differences in similarity (e.g. content or pattern) among these three fragments. Such differentiating however is required to be able to distinguish and rank hits in the result set. Using our multi-level indexing and retrieval approach for the clone search problem we deploys two (or more) indexes at the same time, with each index capturing specific types of similarity (i.e., content or pattern).

5. SECLONE RANKING MODEL

A contribution of our research is that it addresses the ranking of clone search result sets using Information Retrieval (IR) models. The ranking model determines the position of hits in the result set. The position is based on their degree of similarity represented by the pair $\langle query, hit_i \rangle$.

5.1. Ranking approaches

As discussed earlier, the hash values of the encoded code patterns constitute the basic entities within our search space. For our ranking model, any code fragment that shares at least one hash value with a given query will be considered for the ranking. Our ranking approach is based on two similarity models that have been widely used in IR [MAN08]: (1) Jaccard similarity coefficient and (2) the vector space model with cosine similarity.

5.1.1. Jaccard coefficient

The Jaccard similarity coefficient is a widely used set theory function for content matching and measuring the semantic similarities. We calculate the semantic resemblance of two blocks based on their shared content (e.g., lines), regardless of their order. Our ranking model measures the content similarity of two code fragments using the numerical output of the Jaccard coefficient. We denote s_1 and s_2 as the sets which contain hash values that belong to the search query fragment (s_1) and the matched fragment (s_2). Both s_1 and s_2 neither contain duplicate instances nor do they preserve the ordering among entities, due to our non-positional index approach.

$$J(s_1, s_2) = \frac{|s_1 \cap s_2|}{|s_1 \cup s_2|}$$

5.1.2. Vector space model

In addition to the Jaccard coefficient, we also take advantage of the vector space model (VSM) for the ranking of result sets. A key benefit of VSM is that it provides additional flexibility during ranking compared to the Jaccard coefficient. Using VSM, code fragments are represented as vectors of frequency values. Entity frequency can be used to discriminate among entities' contribution by considering both their local and global popularity (oc-

currences). The relevance is expressed as the similarity/distance between a pair of vectors ($query, hit_i$). Similarity is calculated using the cosine similarity function that measures the angle between participating vectors.

$$cosine_similarity(\vec{s}_1, \vec{s}_2) = \frac{\vec{s}_1 \cdot \vec{s}_2}{|\vec{s}_1| |\vec{s}_2|}$$

5.1.3. Weighting factors

Our $|x|$ -dimensional search space consists of code fragments presented as vectors, $\vec{s}_i = \langle h_1, h_2, h_3, \dots, h_x \rangle$, with h_x being the weight (frequency) of an encoded code pattern x . While the local frequency captures the number of occurrences of an encoded code pattern within a particular code fragment, the global frequency represents the total number of code fragments with at least one occurrence of the pattern. Our approach support different models to calculate these local and global frequencies and weights of an entity x within a code fragment i . For example, a combination of l local frequency (Table 5) and t global frequency (Table 6) leads to the well-known IR tf-idf model. Having several ranking options allows us to configure the weights at run-time for different ranking context and to study their effect on the overall clone search performance.

Table 5. Weighting support for local frequency (b_1 parameter)

Function Name	b_1 parameter value	Formula
Boolean	b	$\begin{cases} 1 & \text{if } lf_{x,i} > 0 \\ 0 & \text{otherwise} \end{cases}$
Natural	n	$lf_{x,i} = local\ frequency_{x,i} $
Logarithmic	l	$1 + \log(lf_{x,i})$

Table 6. Weighting support for global frequency (b_2 parameter)

Function Name	b_2 parameter value	Formula
No	n	1
Simple	s	$gf_x = global\ frequency_x $
IR idf	t	$\log\left(\frac{N}{gf_x}\right)$

5.2. SeClone search schema

As previously discusses search schema (Figure 2) in SeClone allows for the configuration of the model properties. The schema includes configuration for (1) the pre-processing and creation of indices for the retrieval phase, (2) the ranking approach, (3) local frequency function, (4) global frequency function, and (5) additional information such as normalization and size comparison functions.

The first parameter of our schema defines the overall ranking approach (Table 7), which can be a variation of cosine similarity, Jaccard similarity, or a combination of both. Furthermore, b_1 and b_2 refer to the local and global frequency functions being used (see Tables 5 and 6). If the Jaccard coefficient is used, only the Boolean local frequency is applicable for b_1 ; and b_2 , b_3 and b_4 will not affect the final result and are set to n (none) to ensure conformance with our schema template. Additionally, we consider the size resemblance between the query and the matched code fragment, which is denoted by b_4 . This option is only applicable for the VSM scoring model.

Table 7. SeClone scoring schemata (α parameter)

Function Name	α parameter value	Formula
Jaccard coefficient	j	$J(s_1, s_2)$
Cosine similarity	w	$\text{cosine_similarity}(\vec{s}_1, \vec{s}_2)$
Cosine Similarity augmented with Size similarity	c	$\text{cosine_similarity}(\vec{s}_1, \vec{s}_2) + b_4(\vec{s}_1, \vec{s}_2)$

Table 8. SeClone size functions (b_4 parameter)

Function Name	b_4 parameter values	Formula
Jaccard coefficient	j	$J(s_1, s_2)$
Simple	s	$ s_1 - s_2 $ where s_i is a bag
Naïve	n	$\begin{cases} \frac{1}{ s_{hit} } & \text{if } s_{hit} > 0 \\ 1 & s_{hit} = 0 \end{cases}$

The size functions (boosters) supported in SeClone are summarized in Table 8. Our search schema also supports normalization of relevance scores, which is denoted by b_3 . Available normalization functions are n (none) and c (cosine):

$$\text{cosine normalization} \quad \frac{1}{\sqrt{\sum_{y=1}^x h_y^2}}$$

In summary, our search schema configures both the retrieval and ranking of SeClone. For example, the *c.ltcj.l1.m3* schema denotes that SeClone uses the cosine similarity scoring schema which is augmented with the Jaccard-based size function (the size booster in this context) to create an IR like *tf-idf* weighting by using cosine normalization function. The indexing is based on single line hash values of Type-1 clones and 3-line hash values of encoded code patterns where only method names have been preserved.

6. DATA CHARACTERISTICS STUDY

In the previous sections, we reviewed our clone search model, SeClone. As we are approaching the actual performance evaluation phase, several issues related to our indexing heuristics can threaten the success of our research. In this section, we conduct a set of preliminary studies to acquire the required insight for the final large-scale performance evaluation phase.

These threats include: (1) the ability to perform clone search with near real-time response time (latency time ≈ 100 milliseconds or less), given outliers, retrieval granularity, and index growth rate of the data, and (2) the ability to maintain reasonable precision of the search result due to potential collisions in our hash function values. For us to evaluate these threats, we conducted a study on the characteristics of data being searched. Such study requires a representative dataset which reflects real data and is large enough to reduce any potential bias within the dataset. For our preliminary study we adopted the UCI dataset [UCI10], which covers over 18,000 Java open source projects extracted from online repositories on the Internet.

6.1. Effect of search granularity on clone search latency times

In the first part of our data characteristic study, we analyze the effect of different search granularities on response times to (1) determine if fine-grained granularities (e.g., single line) are actually practical for real-time clone search over large amounts of data, and (2) estimate the increase in the response time by reducing the granularity. We address these questions by first analyzing the number of retrieved entities (matches) for each element of a query. This analysis provides us with some insight on upper and lower response time boundaries. We observe and compare the worst-case scenarios with respect to the number of matches for both of our two predefined index granularity levels (single and three-line granularity). We first group source code fragments within the dataset in chunks of three lines, with each Third Level Similarity (TLS) group denoting a set of potentially similar three-line code fragments (code clone) where all fragments match the same encoded code pattern. We then repeat the same study for the First Level Similarity (FLS) based on pattern similarity at a single-line granularity.

For creating the dataset, we extract ~ 300 MLOC of non-distinct source code lines, which provides us with a sufficiently large dataset to reduce any potential bias in the data. From this dataset, we then generate 30 million unique TLS groups, covering 71 million distinct lines of source code within method blocks. In our index, each TLS group refers to all occurrences of the same three-line code fragment in the whole repository.

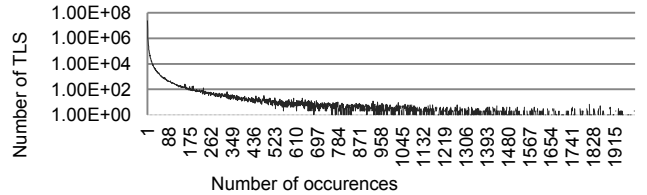


Figure 5. Occurrence frequency distribution for the 3-line (TLS) encoded code patterns

Table 9. TLS and FLS characteristics

Property	Value	
	TLS	FLS
Number of encoded code patterns	30,232,018	7,606,433
Total number of distinct lines	71,911,376	71,911,376
Number of single-member encoded code patterns (one occurrence)	22,824,697	4,770,010
Largest group size (the pattern with most occurrences/members)	1,048,575	2,937,700
Average occurrence frequency	2.37	9.45
Standard Deviation occurrence frequency	293.23	1898.75

Evaluating the index granularity allows us to observe the number of occurrences (including average, min and max) for each encoded code pattern captured by a TLS group. This is of interest, since fewer occurrences will result in a shorter response time. The first observation from this study (Figure 5 and Table 9) is that almost all TLS groups contain less than 2,000 occurrences (instances)

and only a few outlier patterns, 1,220 out of the 30M (0.004%) patterns had more than 2,000 occurrences. Our study shows that the three-line granularity tends to produce large numbers of small groups and very small numbers of large groups. On average, each TLS group (code pattern) has 2.37 occurrences. However, if we exclude patterns with only one occurrence and outliers (with more than 2000 matches), this average would go up to 5.25. From our analysis, we were able to conclude that three-line granularity is practical for real-time clone search, as long as outlier patterns are handled, since it is only for these few outliers that the response time will degrade considerably. Our analysis shows that using TLS, patterns typically occur in small-size groups (on average around 5 members). This is an important observation for our real-time search context since given the small group sizes and our hash-based indexing approach, queries will only be compared against a small number of candidates at run-time.

In addition, similar to our three line similarity (TLS) index experiment, we also studied the distribution of patterns using a single-line level granularity (FLS) index. This experiment actually showed some unexpected differences between the two granularities. For the FLS, the number of outliers (patterns with more 2,000 occurrences) is considerably larger than for the TLS's. This observation is further supported by data in Table 9, which shows that TLS distributes the candidates into 3.9 times more groups, while its group size can be 6 times smaller than the FLS's group size. Moreover, outliers in the FLS index tend to be much larger when compared to the TLS index. Given that the ranking at the group level has a computation complexity of $O(n \log n)$, where n corresponds to the group size, n has a direct effect on the response time. Our study also reveals that while both TLS and FLS are applicable for real-time search, TLS can outperform FLS granularity by a factor of 6.

6.2. The outlier patterns

Outliers often introduce threats to the quality and non-functional performance of search approaches. For example, in text retrieval research, outliers known as stop words are typically eliminated as part of a pre-processing step. As our previous study already showed, while we might only deal with a very small number of outlier patterns (patterns with more than 2000 occurrences) in our dataset, these outliers can have a significant effect on the overall performance of our clone search approach. In order to be able to mitigate this potential threat, it is necessary to identify and study these outlier code clones in more detail. For example our study showed that there exists a three-line pattern with more than one million occurrences (Table 9). If an outlier pattern occurs in the search result set, the ranking algorithm will have to evaluate and rank all occurrences, potentially slowing down the search by a factor of 1000 compared to non-outlier searches (Table 9). For this reason, we further analyze the source code matching these outlier patterns to observe what kinds of programming tasks are associated to the outliers. When analyzing the TLS patterns, we observed

that only 1,220 out of 30 million TLS groups (three-line code patterns) contain more than 2,000 pattern occurrences.

Examples of top 10 outlier patterns are shown in Table 10). Some of the observations from our study are: (1) members of outlier pattern #3 belong to one of the largest open source projects in the dataset (gov.nih.ncgc), which is related to genomics and contains very large files containing these pattern instances. (2) Code fragments in the outlier #6 belong to classes related to the initialization of Graphical User Interfaces. (3) Outlier pattern #8 occurrences can typically be found within extraordinarily large java classes (larger than 10K LOC). The examples in Table 10 illustrate that, similar to the other search domains, outliers in clone search can be also discarded because they are not associated with vital programming problems. Nevertheless, we do not exclude them in our further performance evaluation studies (in this paper) to ensure unbiased and repeatable results.

Table 10. The outlier code patterns

Rank	Number of Occurrence	Pattern Title	Sample Code
1	1304840	Local getter	method() {return variable;}
2	636846	General Setter	method(type arg) { this.variable = arg;}
3	445552	Unknown	s.addToWellOneBased(... new WellComponent(... l.getCompound(..., ...));
4	246082	General getter	method() { return variabile.property;}
5	239604	Local setter	method(type arg) { variable = arg;}
6	124836	Consecutive new	jEdtTest = new JEditorPane(); lblToken = new JLabel();
7	124693	Variable&null	type var1 = null; type var2 = null;
8	115230	Consecutive case	case 'value': case 'value'::
9	100900	Case&return	return "Mountain"; case TYPE_GAS: return "Gas";
10	72842	Throw&new	method(...) { throw (new type());}

6.3. Index size growth rate

Retrieval systems such as [BRI98] keep their indices in main memory, rather than swapped to a disk, in order to reduce latency times when accessing them. In most text retrieval systems [BRI98], the approximate index size is known in advance, as it is directly related to the data characteristics in the domain of discourse (e.g., natural languages). However, for the clone search problem, data characteristics have not yet been fully studied, and no data exists on potential index sizes and growth rates as new patterns and occurrences will be indexed. Without this prior knowledge it is very difficult to determine and allocate in advance appropriate memory resources for creating and storing indices. As we need to know such information for a proper large-scale performance evaluation, we study them as part of our preliminary analysis.

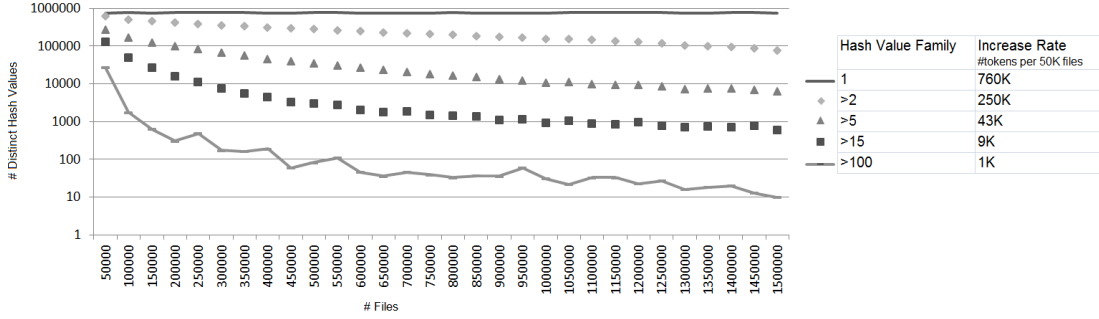


Figure 6. Analysis of the increase rate of new hash values (TLS hashes) per file. Patterns are categorized based on total # of occurrences per hash value.

For a hash table-based indexing system, total memory consumption can be estimated based on: (1) the number of distinct hash values being indexed and (2) the number of pointers required for the repeated hash values. Given that no prior information is available on potential growth rates, we studied the effect of repository size on the index growth rate. We in particular study how different pattern categories (and their indices) evolve as the repository size increases.

For this analysis, we incrementally increase our dataset by adding chunks of 50,000 source code files to the repository. We evaluate the index increase rate for each pattern group, which is summarized in Figure 6. The analysis shows that for popular code patterns (with at least 2 occurrences), the growth decreases over time. This was expected, as the code base being indexed, the likelihood that the same code fragment has already been indexed increases. However, our studies also show that the growth rate for uncommon/unique code patterns remains stable. That is, each chunk of 50K files will introduce a similar number of code patterns that are not cloned and remain therefore unique.

Our study provided us with insights on index growth rate. Finally, using the increase rate table in Figure 6, we can now estimate the index growth via the number of distinct hash values and possible pointers (duplicated patterns), to estimate the feasibility and scalability of our search approach and allocate proper memory resources.

6.4. Hash value strength

Hash table based indexing relies on its ability to maintain indices in the main memory to ensure consistent and fast access times. One approach to reduce the memory footprint is by reducing the length of hash codes, as this will directly affect the memory consumption. However, reducing the length of hash codes can potentially lead to the collision (duplication) of indices. In our approach, we opted to use a 32-bit hash code, which is in contrast to other existing work such as Hummel et al. [HUM10], who used 128-bit code for their clone search approach. The use of smaller hash code (32 versus 128 bits) will not only provide (1) a 75% lower memory requirements for the indices, but can also (2) reduce the latency times due to hardware design.

We conducted an experiment to evaluate whether the use of a 32-bit hash value might potentially introduce a threat to the index quality in terms of collisions. For our

evaluation we created 32-bit hash keys for all single transformed source code lines, using our default transformation function and the Java standard hash method for strings. We extracted more than 4 million distinct transformed lines of code and analyzed the possibility of having an ambiguous key that might be used for more than two distinct lines. The result of our analysis showed that for our 32-bit hash function, the error (collision) rate is small, i.e., 0.002%. Given this low error rate and the resulting tradeoff between precision and memory consumption, we can conclude that the 32-bit hash keys can be considered strong enough for indexing source code. In particularly since for our clone search context, scalability and response times are key requirements.

6.5. Summary

The studies in this section provide valuable insights on data characteristics, such as index growth rates and outliers in a real world, large scale data. Contrary to other research domains, these aspects had not yet previously studied for the clone search problem. We presented the results of our analysis for various data characteristics of the UCI dataset [UCI10]. The insights of our studies benefit the prediction of latency times, index sizes, and overall quality of clone search approaches. Furthermore, our observations also support that our proposed approach using a multi-level indexing and retrieval approach should be capable of providing a real-time and scalable clone search.

7. PERFORMANCE EVALUATION MEASURES

With traditional clone detection techniques putting little emphasize on ranking of result sets [WAL03], ranking quality is typically not part of the performance evaluation. This is in contrast to clone search research, which shares many features with the information retrieval domain, including the need for supporting ranking of result sets. We therefore also consider ranking of clone search results as a quality measure and adopt existing quality and performance criteria commonly used by the IR search community for assessing ranked result sets.

7.1. Requirements

Among the main quality criterion used in IR for evaluating the quality of search engines is the result relevancy from a user expectation. That is, a search is considered to

be successful if it locates documents that are not only related to the query, but also meet the end-user expectations [MAN08]. Therefore, only hits (results) that are relevant from an end-user perspective are considered to be true positives. For example, a result returned by the query “Java”, might only be relevant when one considers the user’s expectation, which might be referring either to the coffee concept or the programming language concept. Such relevancy can be measured on a binary scale (relevant vs. non-relevant) or by using a more refined scale, using different degrees of relevancy (e.g., highly relevant, relevant, marginal, and non-relevant).

Benchmarks are required to measure the quality of result sets reflecting the feedback of either users or experts. They constitute the “gold standard” or “ground truth”. A benchmark or test suite includes three major parts: (1) the input data, (2) some queries, and (3) the pre-tagged dataset of relevant items. The dataset also typically contains relevance scores for each query and its input data, with these scores being subjective to the human experts creating the benchmark. In cases when no benchmarks are available, user studies might be performed.

7.2. The measure suite

For the evaluation of ranked result sets in search applications no single measure has been considered to be sufficient (e.g., [LEM11], [KLX12]). For our research, we identified therefore different categories of ranked result set measures to evaluate clone search models. Most of these measures are based on the definitions provided by Manning et al. [MAN08].

Traditional measures. Recall or precision are typically used by the clone detection and search community to evaluate the quality of any unranked result (sets). These measures are widely accepted or used (search community), since they are easy to calculate and interpret. However, they are not capable of assessing accurately the quality of ranked result sets.

IR measures for ranked results. Most IR systems return result sets that contain some true positives (TP) and false positives (FP) within an ordered list. These IR measures evaluate the true positives and their rank (position) in the result set. Furthermore, relevancy degree is exploited by a subset of measures in this category when all true positives are not equal in quality.

Non-functional performance measures. In our research context, non-functional measures need to be considered when evaluating user satisfaction. We are in particular interested in measures scalability and on assessing the ability to provide near real-time services for other applications.

7.3. Measures for ranked result sets

With many traditional measures like precision or recall being designed to evaluate unranked lists (e.g., unordered sets), the IR community has emphasized special measures for assessing the quality of ranked sets. In this section, we introduce measures mostly adapted from the IR [MAN08] community to assess ranked result sets returned by our clone search models.

7.3.1. First False Positive measure

A commonly used evaluation criteria for search engines in the IR domain are the top displayed items (hits) in a result set. Studies in IR have shown that end-users tend to browse only top items in a displayed result set [MAN08]. Furthermore, since search engines typically do not produce 100% precise results (some non-relevant hits might be displayed), search engines are expected to place as many true positives as possible in the highest ranked position of their result set (e.g., top-10). Therefore, the place of the first false positive (FFP) in the displayed result list can be used as a fair measure for evaluating the performance of search engines. For example, given two order result sets R1 and R2, with both result sets containing 10 hits ($R1 = \langle h_1, fp, h_2, h_3, h_4, h_5, h_6, h_7, h_8, h_9 \rangle$ and $R2 = \langle h_1, h_2, h_3, h_4, h_5, h_6, h_7, h_8, fp, h_9 \rangle$), of which nine results are correct hits and one is a false positive (*fp*). While the precision for both results sets is 90% (9 out of 10 hits are correct), the user satisfaction for R2 would be considered higher, since the FFP occurs later in the ranked result set (position 9 in R2 versus position 2 in R1).

Discussion. In clone search, one typically deals with a corpus that contains a significant amount of noise (irrelevant code fragments). For example, in one of our case studies we observed that for some queries only 6 out of ~1.7 million code fragments in the corpus were relevant. Therefore, from a code/clone search perspective, a search approach has to deal with two major challenges: (1) being able to detect the few relevant fragments, and (2) assigning these true positive results a higher priority than the false positives in the result sets. The First False Positive idea provides an easy to use and interpret measure to assess some quality aspects of an ordered result set.

Weakness. Given that the FFP is highly dependent on the data and query characteristics, its applicability to evaluate system performance is often limited. For example, if a corpus contains a skewed dataset with only X true positives for a given query, the best achievable result using this measure is $X + 1$. This becomes an issue particularly in cases where the number X (true positives) varies considerably for different queries. Consequently, the FFP measure cannot be generalized since it cannot be averaged across different queries.

7.3.2. “Precision at k ” measure

Precision at K ($P@K$) is a measure that reports the number of true positives within the hit list (top K), where K can be any positive number to reflect the window size for the assessment. However, window sizes of 10, 20, and 30 are typically used for K s. The value of K is derived by the general rule of thumb from search engine GUI design, where the first page usually shows only the top 10 hits. The measure captures closely the end-users quality perception, since users tend to consider only results on the first result page to be important and consequently are less likely to browse subsequent result pages.

$$Precision_k = \frac{tp_k}{tp_k + fp_k}$$

where tp and fp are limited to the top K hits

This measure is in particular applicable when (1) the total number of relevant results is unknown and therefore

no recall can be computed, and (2) the number of returned items is too large to be fully validated, making the calculation of standard precision measures impossible.

Weakness. While this measure is a good candidate for evaluating search engines, especially when no very strict evaluations (e.g., “first false negative” measure) are required, its major drawback is its dependency on the query. For example, in order to provide a fair evaluation using “Precision at 10” measure, at least 10 actual relevant items must exist in the corpus for all executed queries. Furthermore, similar to the FFP measure, results from this measure cannot be generalized (averaged) across queries.

7.3.3. Normalized Discounted Cumulative Gain measure

The Normalized Discounted Cumulative Gain (NDCG) measure assesses the quality of search engines and their ranking algorithms in terms of being able to assign higher ranks to more relevant true positive answers. NDCG takes into consideration not only the relevance of hits but also the order of the results. Therefore, the measure allows comparing result sets for queries using an existing oracle. These oracles are typically manually created result sets (for each query) which include a list of all possible answers. Moreover, each answer in the oracle must be assigned a relevance score that presents its relevancy to a the query. The oracle captures the best overall achievable result set (including the order of answers), independent of local configurations, search algorithm, and search schema. The measure result is a number which allows comparing different search and ranking configurations.

Details. DCG calculates the discounted cumulative gain achieved using a given search schema for query q when compared to the oracle with its manually assigned relevance scores for the top n hits. The output of DCG depends on the query and available data within the corpus ($DCG \in [0, \infty]$). It is not possible to compare directly DCG results of different queries with each other since the number of positive hits is dependent on the data characteristics of corpus. To overcome this limitation and to be able to compare results, we use NDCG, which is a normalized value of DCG. We first calculate the Ideal DCG (IDCG), which is the highest achievable DCG given the available relevance scores in the oracle. Using DCG and IDCG, we can then calculate the final NDCG value. The function $r(q, i)$ returns the relevancy score based on a given query and a corresponding hit from the oracle.

$$DCG(q, n) = r(q, 1) + \sum_{i=2}^n \frac{r(q, i)}{\log_2(i)} \quad \text{and} \quad NDCG(q, n) = \frac{DCG(q, n)}{IDCG(q, n)}$$

Discussion. Since the output of the NDCG function is normalized, it can now be used for both (1) query comparison and (2) as an overall indicator for the performance of a search engine. The ability to average the measure results provides a concrete single output value for performance comparison purposes. For example, in our studies we use this single output value to compare the performance of different search configurations. The value for the NDCG function ranges from 1.0, for a result set that exactly

matches the oracle, and a minimum of 0.0 for a result sets with no true positive.

Weakness. The measure allows for a fine-grained evaluation of the quality and ordering of result sets, by providing a single value assessment that allows the comparison among different options or configurations of a system. However, the measure is only applicable when fine-grained ordering is important, otherwise measures such as Precision at K are preferred. Applying NDCG is also expensive, since not only all possible answers for each query have to be evaluated manually, but also a similarity score for each answer is to be provided. Nevertheless, NDCG is still considered as one of the state of the art search engine measures in the IR domain.

7.3.4. Mean Average Precision measure

Mean Average Precision (MAP), is a single value measure that has been commonly used to compare different ranking systems. For a single query experiment, the measure will compute the average of all precision at K s, where K s refers to the position of all relevant retrieved items in the result set. For experiments involving more than one query, the output is the average of all queries.

$$AP = \text{Average Precision} = \frac{1}{|R|} \sum_{k \in R} \text{Precision at } k$$

where R is the **set** of all relevant retrieved items

$$MAP = \frac{1}{|Q|} \sum_{q \in Q} AP_q \quad \text{where } Q \text{ is the } \mathbf{set} \text{ of all queries}$$

Weakness. MAP is an essential and low cost measure that does not require the creation of relevance scores (unlike NDCG) and only considers the positions of true positives. However, since MAP does not include relevance scores, it lacks the ability to compare true positives from relevancy aspect. Moreover, it is generally only suitable for queries where a reasonable number of true positives are available; otherwise its output might be biased.

7.4. Measures for highly positive ranked results

Sometime, no or only a few fp are included in the hit list (e.g., top 10). While all hits might be true positives, end-users often rank some true positives higher than others. Assessing this type of ranking requires more precise measures that take also into account the exact order of tps in a ranked result set and compare them against the oracle. Such measures are different from earlier measures introduced in this section (e.g., NDCG), as they evaluate the relative or exact position of all items within the ordered list. Several measures have been introduced to assess this type of ranking performance.

7.4.1. Normalized Kendall’s τ distance

Kendall’s τ measures the dissimilarity of the items’ order against the ideal order [LAP06]. Suppose π and σ denote the ordering of two item sets containing the same items, with N . $S(\pi, \sigma)$ being the minimum number of switches required between adjacent items to make the first ordered list identical to the second ordered list.

$$\tau = 1 - \frac{2 \times S(\pi, \sigma)}{N(N-1)/2}$$

7.4.2. Spearman's rank correlation coefficient

This measure compares the rank of each shared retrieved item among two subject ranked lists, which are denoted by π and σ with the number of items being equal to N .

$$\text{Spearman} = 1 - \frac{6 \sum_{i=1}^N (\pi(i) - \sigma(i))^2}{N(N^2 - 1)}$$

$\pi(i)$ and $\sigma(i)$ are referring to the rank of item i in the corresponding hit list

Discussion. As Lapata [LAP06] pointed out, the main difference between Spearman's and Kendall's measures is that Spearman's measure focuses on the pure rank values, whereas Kendall's measure emphasizes more the relative order of items.

7.5. Non-functional performance measures

For our evaluation of the quality of result sets we also consider non-functional measures that can potentially affect user satisfaction (e.g. the ability to provide real-time services for other applications). We consider for clone search engines are: (1) indexing time, (2) querying latency time, and (3) corpus size. These measures are easy to calculate (automatically) and interpret.

7.6. Summary

Assessing the quality of clone search (models) differs from traditional clone detection. While traditional clone detection approaches deal with unranked result sets where measures like recall and precision matter, they do not consider the order of the results being displayed. This is in contrast to clone search, where, as in other search approaches, the ranking of results (ranked hits) becomes a key quality criterion. While evaluation measures designed for unranked result sets are useful (e.g., precision and recall), other evaluation measures which are developed for ranked result sets must be adopted to provide a more comprehensive evaluation of a clone search model. As part of our research, we selected and summarized several ranked result set quality measures, originally used by the IR community [MAN08], and highlighted their applicability in our clone search context.

8. PREPARATION FOR EVALUATION STUDY

While our preliminary results from the data characteristic study (Section 6) support the feasibility of our solution (run-time behavior), a more detailed performance evaluation study is required. In this section, we discuss details of our evaluation, which takes advantage of the insight from our initial data characteristic study. For the performance study, we deploy a concrete instance (SeClone) of our clone search model and apply it to our source code corpus, which contains source code facts from over 25,000 open source Java projects [KLF12]. The key objectives of this evaluation are (1) to confirm that our model can meet the scalability and fast response time requirements and (2) to compare different search schemata (configurations) available in SeClone. Benchmarks are commonly used approaches to evaluate the quality of search engines. For us to be able to assess different features of our model (SeClone), including both retrieval and ranking, we require a benchmark that meets a set of minimum requirements, including: the corpus (1) should be large enough to re-

duce the effect of individual outliers, (2) contains a set of representative queries (code fragments) to be used as search criteria, (3) includes a sufficient number of relevant Type-1, 2, and 3 clones, and (4) includes fine-grained relevance scores for clones. To the best of our knowledge, there exists no clone search benchmark that satisfies all these requirements. Therefore, prior to our evaluation, we had to create such a clone search benchmark. As part of the benchmark creation, we took advantage of an existing mutation generation framework [ROY09], which we used to automatically generate Type-1, 2, and 3 clones from 50 randomly selected code fragments (query inputs). 50 queries can be considered an acceptable number for a benchmark [MAN08]. For these 50 code fragments, we generated a total of 650 related Type-1, 2, and 3 clones. For the benchmark preparation, we injected not only these 650 clones generated by the mutation framework into our repository (which contains 356M LOC), but also performed an extensive manual inspection of ~80K code fragments for assigning their corresponding relevance scores. We then use this benchmark to assess SeClone's search performance using the five measures introduced in the previous section, while analyzing over 40 different SeClone configurations (search schemata). This evaluation involves 2000 queries (code fragments) for which a clone search was performed, resulting in 117,000 search results (hits). The following sections describe in more detail our evaluation approach, its outcomes, and summarize of our findings.

8.1. Candidate search schemata

SeClone supports hundreds of different configurations through its search schemata. These configurations allow users to specify different search models, indexing granularities, and content transformation functions. From an end-user perspective, selecting an appropriate configuration is the key to meet specific application or end-user needs. In our study, we conduct a detailed analysis of 40 candidate configurations to determine their effect on the quality of the result sets and to provide guidance for end-users when selecting a search schema. In sections 3, 4, and 5 we already introduced in detail SeClone's search schema and its configuration options: (1) parameters (a, b_1, b_2, b_3, b_4) related to the ranking and (2) parameters (t_p, g_p, t_s, g_s) related to the processing of data for indexing and clone analysis. For our experiments, we selected five ranking configurations and eight indexing (analysis) configurations, which provided us with a total of 40 distinct configurations (Table 11).

8.2. Corpus and environment configurations

For the deployment of SeClone, we used a Linux-based system with a 3.07 GHz CPU (Intel I7) and 24 GB of RAM. During our run-time evaluation, a configuration was executed as single process, except for the Java virtual machine processes such as garbage collection. In order to evaluate the scalability, response time, and ranking, and to observe the handling of outliers (noise), a reasonably large corpus was required. For the evaluation we created the IJaDataset, a large multipurpose source code data set.

Table 11. Selected SeClone search schemata for the evaluation phase
(SeClone Search Schema: $a. b_1 b_2 b_3 b_4. t_p g_p. t_s g_s$)

The first parameter group (ranking) $a. b_1 b_2 b_3 b_4$	×	The second parameter group (indexing) $t_p g_p. t_s g_s$	=40 search schemata
(A) <i>c.ltcj</i> (Cosine similarity augmented with Jacard size similarity using tf-idf like frequency)		<i>c1.m1</i> (1)	
(B) <i>c.nscs</i> (Cosine similarity augmented with Simple size similarity + natural frequency)		<i>c1.m3</i> (2)	
(C) <i>j.bnn</i> (Jaccard coefficient similarity approach)		<i>l1.m1</i> (3)	
(D) <i>w.ltcn</i> (Cosine similarity using tf-idf like freq.)		<i>l1.m3</i> (4)	
(E) <i>w.nscn</i> (Cosine similarity using natural frequency)		<i>w1.m1</i> (5)	
Total 5		<i>w1.m3</i> (6)	
		<i>x1.m1</i> (7)	
		<i>x1.m3</i> (8)	
		Total 8	

The dataset contains Java source code data crawled and downloaded from major open source code repositories (e.g. Sourceforge) [KLF12]. We performed several data cleaning steps, such as: (1) removal of all non-Java source code and duplicate Java files, (2) using a Java parser, we detected and removed all unparseable files, and (3) we identified and excluded Java interfaces, since these Java interfaces do not contain any significant code.

The most recent version of the IJaDataset (Version 2.0) has been updated with data crawled in 2012 and covers approximately 25,000 projects and includes Java classes without package specification (default package) [KLF12]. The dataset is based on source code files that were downloaded from SVN, Git, and CVS repositories from SourceForge and Google Code. To remove high-level duplications in the dataset, only one Java File is selected for each available class name identified by its fully qualified name (FQN). During the filtering of duplications, we were biased toward files that appeared in the "trunk" directory. The crawled data (with duplicated files) initially included 12 million files, but were reduced (through the filters) to 3 million files (2.7M regular Java class source code files and 140K files with default package). We then successfully indexed all 356M LOC in the IJaDataset (Version 2.0) with SeClone to create a single, searchable corpus. To the best of our knowledge, this represents the largest *inter-project* Java data set (based on real source code) that has been used for clone search. The IJaDataset dataset is publicly available for download and reuse (<http://secold.org>).

8.3. The benchmark

A high-quality benchmark for clone search should not only include queries and their correct answers, but also a variety of clone types (specifically Type-3 clones) for *each query*. Having such a rich benchmark provides not only the basis for evaluating our core SeClone search engine, but also for evaluating its ability to rank result sets and detect Type-3 clones. Using the mutation framework introduced in [ROY09], we created our initial benchmark using 50 code fragments (queries) and their mutants, which correspond to Type-1, 2, and 3 clones. We selected a mutation framework configuration that automatically

generates 13 clones (4 Type-1s, 3 Type-2s, and 6 Type-3s) for each query. Table 12 summarizes the clones. In case of code insertion when generating Type-3 clones, the mutation framework uses random code snippets available in its corpus. The generated clones were then included and indexed as part of our SeClone corpus. Using this mutation approach provides us with a known minimum number of true positives. Therefore, we are able to (partially) measure the recall in addition to the other precision-like measures. It should be pointed out that since the corpus contains millions of indexed lines of code, SeClone will not only detect and retrieve the seeded clones, but also may include other (positive) clones in the search results.

8.4. Assignment of relevance scores

When evaluating the performance of search engines, solely measuring true positives is not sufficient. In addition one has also to consider the relevance (score) of the returned search results (hits) with regard to a given search query. For our evaluation we therefore assigned scores (in the range of 0 to 5) to indicate the relevancy of a hit and its search query. A score of 0 reflects no relevancy (false positive), and scores between 1 and 5 denoting that a hit has some degree of similarity (true positive $\langle query, hit \rangle$ clone pair). Increasing scores indicate higher levels of similarity, with a score of 5 being an exact (Type-1) match. As part of creating our benchmark we have initially assigned such relevance scores to the 650 cloned fragments that were generated by the mutation framework and the corresponding search queries for retrieving them.

Given the size of our corpus (25,000 projects and 356 MLOC), there is a good chance that other true positives might be reported during the evaluation process. The relevancy of detected and reported clone pairs therefore not only depends on the returned injected clones but also on the non-seeded and reported clones, which have to be considered as part of an overall evaluation. We therefore in addition (to the seed clones) (1) manually evaluated all reported hits to determine if they are actual true or false positives and (2) assigned the proper relevance scores using a predefined guideline (Table 13).

Since it is both impossible and unnecessary to consider all potential hits retrieved for each query in the benchmark (a query might return thousands of hits), we decided to consider only the top K hits. While it is common best practice in the IR and search community to consider the top 10 hits, we decided to increase the evaluation scope by including the top 60 hits. This extended evaluation is motivated by the characteristic of our corpus, considering the fact that we have generated and included at least 13 controlled, true positives (clones generated by the mutation framework) for each of the 50 queries.

As part of our evaluation, SeClone reported for the 2000 executed queries a total of 117K hits (clone results) when considering the top 60 criterion. We used some basic heuristics (e.g., hit size and keywords) to automatically identify some of the false positives and eliminate them from the manual analysis process. Using these heuristics, we were able to eliminate 37K false positives that no longer required a manual inspection and scoring. We

then manually assigned relevance scores to the remaining 80K results (32K distinct $\langle query, hit \rangle$ pairs). Table 14 summarizes the details of this manual assignment of relevance scores for which considered both syntactical and semantic similarities. The manual relevance score assignment has been done in 3 months by the first author.

Table 12. Available clones for each query in the benchmark

ID	Description (changes comparing to the query)	Clone type	Our relevance score
1	no change	Typ-1	5
2	changes in whitespace	Typ-1	5
3	changes in comments	Typ-1	5
4	changes in formatting	Typ-1	5
5	semantic renaming of identifiers	Typ-2	4
6	arbitrary renaming of identifiers	Typ-2	4
7	arbitrary change of an literal	Typ-2	4
8	replacement of identifiers	Typ-3	3
9	small insertion within a line	Typ-3	3
10	small deletion within a line	Typ-3	3
11	insertion of one or more line	Typ-3	2
12	deletion of one or more line	Typ-3	2
13	modification of entire line	Typ-3	3

Table 13. Guidelines for assigning relevance scores

The assigned score	Scoring guideline
0	Non-relevant
1	Relevant (partial similar under Type-3)
2	Relevant (Type-3 with modification of few lines)
3	Relevant (Type-3 with one line different)
4	Highly Relevant (Type-2)
5	Highly Relevant (Type-1 / exact)

Table 14. The evaluation steps and hits manual investigation details

Property	Value
Total search schemata	40
Total benchmark queries	50
Total querying experiments	2000
Result set limit	Top 60
Total retrieved hits	117K
Total number of hits which are automatically ignored using heuristics	7.7K (size heuristic) 28K (keyword heuristic)
Total number of hits which are tagged manually	81K (32K distinct $\langle query, hit \rangle$ pairs)
Breakdown	
<i>Score</i> <i>#hits</i>	
0	34K
1	14.9K
2	3.6K
3	15K
4	4.9K
5	8.8K

9. PERFORMANCE EVALUATION

In this section, we present the results of our performance evaluation study. We conduct this study to answer two major concerns with regard to the applicability of SeClone:

G1. We study if SeClone can be used for *interactive search scenarios* with large-scale corpus. 100 milliseconds is the de facto response time for interactive search [BAS13]. Therefore, as the first step, we study scalability and response time of SeClone for the candidate search schemata (Table 11).

G2. The second goal of our study is to evaluate the performance of SeClone in terms of ranking. We study whether SeClone can successfully place the known positive answers, from the benchmark, at the top of the ranked lists before the other less similar answers (e.g., false positives). To evaluate the ranking performance of SeClone, we use five applicable measures from the measure suite introduced in Section 7. We present the results for G1 and G2 in Sections 9.1. and 9.2. consecutively.

9.1. Scalability and response time

One of the key requirements for SeClone, being a specialized search engine, is the need to be scalable and to provide search results in near real-time (i.e., 100 milliseconds [BAS13]). In what follows, we discuss SeClone’s run-time and scalability performance based on the execution of our benchmark queries. For the analysis, we consider clone lookup times, ranking, and sorting as the total response time, which is reported in milliseconds.

It should be noted that to deploy the SeClone server application and its indices, SeClone requires 10 minutes for the incremental indexing of the encoded code patterns covering the 356M LOC (3 million Java files).

Figure 7 summarizes the response times that we observed for the 50 queries executed for each of the 40 schemata (Table 11). The schemata are identified by their short names that are highlighted in bold in Table 11. For example A1 denotes *c.ltcj.c1.m1* schema. In this study, we found five different configurations (out of forty candidates) that are both scalable and real-time, i.e., ~ 100 ms, even for our ultra-large scale corpora. Users can use these configurations for clone search and get scalability and near real-time response time experience, e.g., A4 (*c.ltcj.l1.m3*) and C4 (*j.bnnn.l1.m3*) that are tagged in Figure 7.

Next, we examine the results to find the reason behind the success or failure of some of the schemata. We observe that all of the successful schemata use *l1.m3* as the indexing configuration. Therefore, first, we analyze the role of indexing configuration on response time. As we discussed in Section 6, a successful combination of *index granularity* and *transformation function* can achieve real-time response time only if it distributes well the indexed entities across the index. Our analysis confirms our earlier discussion. Both *index granularity* and *transformation function* can significantly affect response times. Mann-Whitney U test (a non-parametric test) shows that a statistically significant ($p - value < 0.05$) improvement could be obtained where *l1.m3* is used as the primary and secondary indices. Similarly, we compared the response times of the two major ranking models, i.e., Jaccard and VSM. We observe that the choice of ranking model does not affect response time significantly in the context of clone search.

Finding. Indexing configuration (i.e., granularity and transformation function) affects significantly the response time in our research context.

Finding. Ranking model (i.e., Jaccard or VSM) does not affect significantly the response time.

Finding. G1. SeClone achieves both scalability and near real-time response time by certain schemata.

9.2. Result set quality

In this section, we study the performance of SeClone from the quality point of view. This study addresses the second goal of our performance evaluation analysis (G2). In general, we are interested to observe whether SeClone (1) identifies the known true positives in our benchmark, and (2) places the true positive answers at the top of the result sets. These two questions can be mapped naively to the traditional recall and precision concerns respectively.

First, we provide a brief summary of SeClone performance in the context of traditional recall and precision in Section 9.2.1. However, as discussed earlier, the concrete evaluation approach should be based on the concept of ranking. Therefore, we report the details of our actual evaluation study on SeClone ranking in details after the brief summary in Section 9.2.1.

9.2.1. Can SeClone detect the true positive answers?

In this section, we provide a naïve summary of SeClone performance as we were concerned whether SeClone can detect any of the known true positive clones. We are interested to observe whether the SeClone retrieval model can really achieve high recall.

We observe that there are 27 schemata of SeClone, out of 40 tested schemata in Table 11, that detect all of the 13 known positive answers (Table 12) from the benchmark, including the Type-3 clones. From a benchmark-based evaluation point of view [BEL07], SeClone achieves 100% recall. Our initial analysis also shows that for 96% of queries, some of the schemata with high recall also achieve 100% precision for top K hits within the result sets, with K being equal to the number of expected positive answers from the benchmark. These results are promising, specifically (1) on a noisy ultra large corpus (2) with an approach that is not aware of the positional information of source code. However, these numbers should be interpreted carefully due to our benchmark-based evaluation approach, as discussed by Bellon et al. [BEL07]. We report the details of our observation in the next sections.

9.2.2. First False Positive

In the previous section we reported that SeClone achieves high precision and recall. However, it does not mean SeClone is the perfect clone search model. In the following, we report the details of our analysis and the observations that we made showing both pluses and deficiencies of SeClone. Our major concern is to observe whether the fast schemata, that are shown to perform in real-time in Section 9.1, are amongst those producing high quality ranked result sets. In other words, is there any configuration of SeClone that performs well from

both quality and response time points of view? In the following, we refer to the certain schemata of SeClone that achieve both scalability and near real-time response time (Section 9.1) as *target schemata*, e.g., A4 (*c.ltcj.l1.m3*) and C4 (*j.bnnn.l1.m3*).

Figure 8 provides a summary of the result for the First False Positive (FFP) measure. The observation is based on the FFP values for all queries across all 40 search configurations (schemata). The results show that the first false positive appears on average at the 25th position for most schemata. Among the 40 schemata, two of them considerably outperform the others by having first false positive at position 30 on average, whereas 9 of the 40 schemata perform poorly. *c.ltcj.l1.m3* (A4) and *j.bnnn.l1.m3* (C4) outperform the other schemata, even the other target schemata, i.e., *w.ltcn.l1.m3* (D4) and *w.nscn.l1.m3* (E4). Therefore, regarding the earlier question, there is no schema better than SeClone real-time schemata from FFP point of view. We consider A4 and C4 as the *surviving target schemata* in our further analysis. Note, both schemata detect all of the 13 known positive answers from the benchmark. Our observation also shows that SeClone finds and ranks further positive answers beside the benchmark’s 13 positive answers at the top of the result set, achieving 30 (average) for FFP, e.g., using A4 schema.

9.2.3. Precision at K

In the FFP section, we studied the position of the first false positive answer. FFP is a conservative measure. To provide a higher level overview of SeClone performance, we consider Precision at K measure (P@K). We study 7 different scenarios: K = 10, 15, 20, 30, and 60. The motivation for evaluating these different K values was to provide an analysis of SeClone’s performance as K increases. We limited the K value to a maximum of 60, since we only tagged the top 60 hits during our relevance score assignment step. We observe that for precision at 10 and 15, SeClone achieves 100% precision (for 48 out of 50 queries) for both K ranges. As expected, the precision values drop as the K values increase to 60 (Figure 9). The major reason for this drop in precision is mainly related to data scarcity. This is partly caused by our benchmark, since we generated through the mutation framework (and seeded afterwards) only 13 confirmed clones for each query. As a result, the precision at K values higher than 13 depends on the actual data availability in our corpus, which is non-deterministic, in particular given the size of the corpus and the differences among queries.

An interesting observation can be made for schemata such as *c.ltcj.l1.m1*, when the second index uses the *m* transformation function at the single line granularity level (i.e., *m1*). These search schemata actually achieve the highest median value, which can be explained by the fact that for such a fine-grained index, the engine was able to detect a large number of true positives to achieve higher recall. However, the improvement is not statistically significant comparing to our *surviving target schemata* from the FFP study (i.e., A4 and C4).

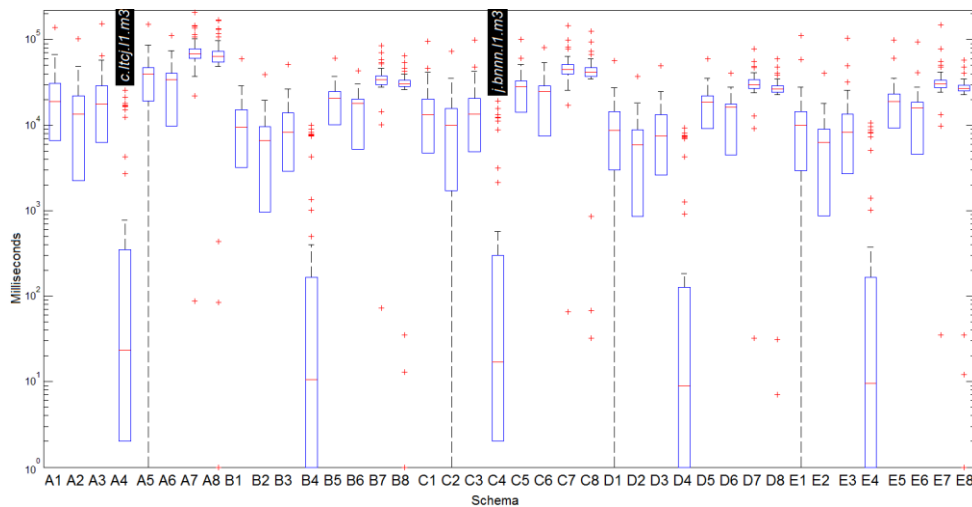


Figure 7. SeClone response time using a 356M LOC corpus

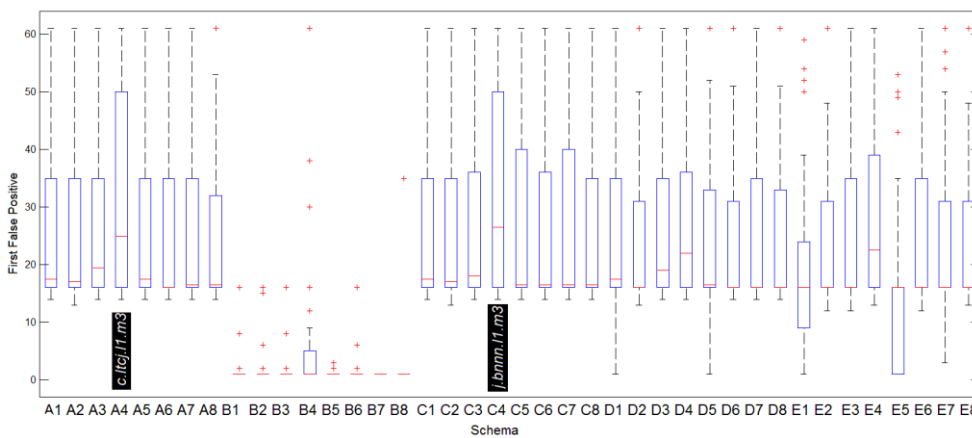


Figure 8. First False Positive result across all configurations using a 356M LOC corpus

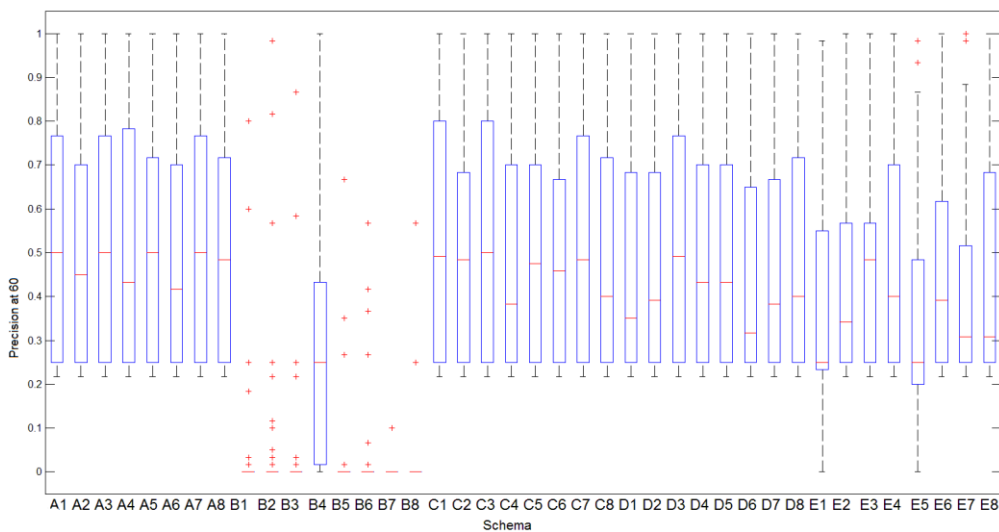


Figure 9. Precision at 60 result across all configurations using a 356M LOC corpus

9.2.4. MAP

Our first two studies (i.e., FFP and P@K) evaluate the performance of SeClone from the precision point of view. We observe that for a reasonable K value (i.e., 15) with regard to our benchmark characteristics, SeClone achieves complete precision for P@K. However, as K increases the precision value decreases. Our initial observation highlights that the drop is due to data scarcity. However to answer this concern concretely, we further study the Mean Average Precision (MAP) measure. MAP is a single value measure typically used in the IR community to evaluate ranking systems. For a single query experiment, the measure will simply compute the average of all Precision at K s where K s refers to the position of all retrieved relevant items in the result set. MAP is useful when the degree of *similarity* (relevance score) of true positives is not of importance. We observe that both of our *surviving target schemata* (i.e., *c.ltcj.l1.m3* and *j.bnnn.l1.m3*) achieve almost complete MAP value, i.e., 0.98. Achieving a MAP close to 1 means that if there is a positive answer within our benchmark, SeClone has placed the TP almost at the top in most of the cases. This supports the idea that the drop in P@K for large K s, is due to data scarcity, not SeClone malfunctioning. Note, we also observe that there is also no other schema that significantly achieves a higher MAP value than our *surviving target schemata*.

9.2.5. Normalized Discounted Cumulative Gain

In our previous studies, we observed that there are two schemata of SeClone that not only provide real-time clone search but also find the true positive answers and place them at the top of the ranked result set. However, there are further performance factors to be considered for a successful clone search model. Similar to the other search domains, all true positive answers in clone search are not equally relevant to the query. In our benchmark (Tables 12 and 13), we define the relevance degree of answers to the query based on clone type and the degree of dissimilarity. To evaluate SeClone *ranking* for applications where the relevance score of true positives is emphasized, we used the Normalized Discounted Cumulative Gain (NDCG). We observe that *x1.m3* index configuration (e.g., A8 schema) achieves the best NDCG in our study. However, the improvement is not statistically significant comparing to our *surviving target schemata*. Our NDCG study supports and confirms that both of our target real-time schemata (i.e., *c.ltcj.l1.m3* and *j.bnnn.l1.m3*) perform well by achieving 0.9 out of 1 NDCG in the worst case.

9.2.6. Normalized Kendall's τ distance

In the NDCG study, we observed that SeClone can produce high quality ranking in terms of relevancy. However, it does not mean that SeClone is a perfect clone search model for ranking clones. In this section, we discuss the limitations of SeClone in ranking clones using Normalized Kendall's τ distance. We use the Kendall tau measure, since it provides a fine-grained comparison of highly positive result sets based on their relative order.

Since SeClone search schemata rank result sets based on their content similarity, Type-1 and Type-2 clones

(similarities) are consistently placed in the correct relative order and position within the result sets. However, for Type-3 clones, the relative position (compared to Type-1 and 2 clones), depends on the dissimilarity between the clones and the query fragment. Using Kendall's τ , we study how close our ranking approaches can match an optimum ranking (e.g., Table 12), in the exact order. The outcome value for Kendall's τ can be between -1 and 1. The evaluation of our recommended schemata (*c.ltcj.l1.m3* and *j.bnnn.l1.m3*) shows that SeClone is neither poor (i.e., -1) nor perfect (i.e., 1) in this context. However, SeClone is closer to 1 than -1. Figure 10 shows the summary of the observation. Although the median values for both schemata are close, the Jaccard coefficient search schema C4 (*j.bnnn.l1.m3*) outperforms the VSM-based schema by providing consistent (better) ranking results. Nevertheless, the difference between the two configurations is not statistically significant.

In summary, in this study we showed the promising achievements of SeClone using FFP, P@K, MAP, and NDCG measures. However, we observed that SeClone is not the perfect clone search model using the Kendall's τ observation and can be improved, if an exact ranking as Table 12 is expected.

Finding. G2. SeClone detects positive answers and ranks them at the top of the result set before the other irrelevant answers with an acceptable performance, i.e., 0.9 NDCG.

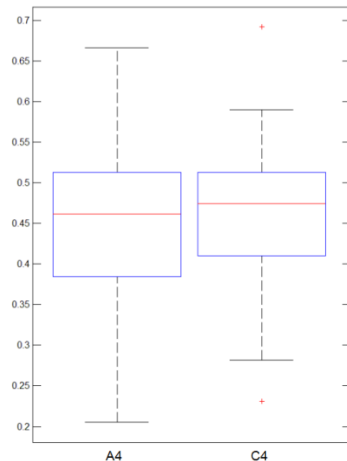


Figure 10. Kendall's τ distance study on the two surviving schemata

9.3. Discussion

We observe that all of the 9 schemata belong to the B series underperform the others significantly. Common to all of the schemata belong to B series is the *simple size similarity booster*. This function simply assigns higher rank to hits that are having the same size (i.e., unique patterns) as the query. We have been selected this function for our study based on its success in our earlier preliminary studies. In our preliminary studies on single-project datasets, the function performance was acceptable due to the characteristics of small-size datasets (e.g., fewer pitfalls). In-

terestingly, as we observe in this study, the function fails significantly for clone search on a noisy corpus. This observation is interesting as it highlights the challenges of clone detection and search on ultra-large noisy datasets, i.e., a heuristic that works well for small-size dataset, is not necessarily suitable for large-scale clone search and detection.

10. THREATS TO VALIDITY AND LIMITATIONS

The clone search model proposed in this research can be adapted for different application contexts which require scalability, fast response time, ranking, and Type-1, 2, and 3 detection (e.g., [LEM11]). This section summarizes some ongoing concerns that must be taken into consideration before adopting our clone search model.

Data characteristics study. Our data characteristics studies provide essential insights on the data used in our research and application domain (e.g., corpus growth rate, data outliers, and the strength of the hash function). However, the observations reported from our studies depend on three major factors: (1) the corpus being searched, (2) the data granularity used for the study, and (3) the selected encoded code patterns. Although we tried to consider a representative dataset for our studies, all conclusions drawn from our case studies remain highly dependent on the corpus. For example, when using a dataset from industrial or closed code systems, the conclusions will most likely differ, since the characteristics of the code might differ. Furthermore, our studies are limited to Java source code. For the granularities, our results are limited to line-level clone detection, and the conclusions are not generalizable for other granularity levels. Finally, we have selected encoded code patterns that will result in high recall. Achieving high recall helped us to study the worst-case scenarios for our retrieval and ranking steps, as it resulted in a large number of candidates to be ranked. Therefore, the observations are not generalizable, as is, to the other encoded code patterns that emphasize on other requirements.

Performance evaluation study. Considering our evaluation approach, the quality of our benchmark plays an important role, since it has a direct impact on the outcome of the performance evaluation. Therefore, the following issues must be taken into consideration. Since no other benchmark was available for the evaluation of clone search results and ranking performance, we created our own benchmark using a mutation framework to generate an oracle of known clones. A key challenge, as with any benchmark is, how closely such a benchmark reflects real world data. We address some of these threats by creating a dataset that we believe is representative enough in size (containing 25,000 different open source projects and approximately 356 MLOC). Furthermore, the mutation framework output (our oracle) is injected to our corpus to ensure that a minimum number of clone instances existed for each query and to allow recall calculations. We also identified other possible positive answers from the original corpus beside the injected clones similar to Bellon et al. approach [BEL07]. In an attempt to reduce the subjectivity

during the manual scoring process, the scoring process was made as transparent and objective as possible similar to Bellon et al. approach [BEL07]. We also followed predefined guidelines provided by the mutation framework to setup the scores.

Implementation. We have implemented our clone search models and all of its processing components in Java. While we performed extensive testing of our implementation, we did not consider a formal validation of either the design nor of the implementation (including the programming heuristics).

Limitations. Our study focuses on a clone search model for Java source code. However, support for other programming languages would require typically the substitution of the language parser. While our model can be applied to the other programming languages such as C, its performance might differ significantly. This is due to the fact that our encoded code pattern generation rules have been designed and optimized based on characteristics of Java source code available on the Internet, after a detailed experimental of existing code search query logs analysis (see Appendix).

11. CONCLUSIONS

In this research, we study the potentials of Information Retrieval models for code clone search. Our research presents a clone search model which not only supports scalability (i.e., Internet-scale), short response times (i.e., real-time), and Type-1, 2 and 3 detection, but also emphasizes the ranking of result sets as a key functionality. This ranking is used to place highly similar fragments (hits) higher than other hits within the result set. Our clone search model (SeClone) takes advantage of a multi-level indexing (non-positional) approach to achieve a scalable and fast retrieval with high recall. Result sets are ranked using two Information Retrieval ranking approaches: Jaccard similarity coefficient and cosine similarity via the vector space model. We combined these ranking models with code patterns' (not token) local and global frequencies functions, which can be used to customize the search schemata to specific application requirements.

For the evaluation we created a large corpus (365M LOC). The corpus in combination with 50 sample queries and a total of 650 seeded Type-1, 2, and 3 clones formed our benchmark. This benchmark, including an extensive manual tagging of relevance scores of over 117,000 hits. The benchmark is used to evaluate SeClone retrieval and ranking quality. We selected five quality measures to evaluate and identify schemata, which can consistently outperform others. The analysis showed that SeClone not only scales to very large datasets but also can produce high quality results in near real-time using the identified schemata (*c.ltcj.l1.m3* and *j.bnnn.l1.m3*).

As part of our future work we plan to extend and evaluate our clone search model to token-level similarity search granularity using non-positional indexing. We are also planning to release SeClone as an online clone search engine.

REFERENCES

- [ABR79] P. S. Abrams and J. W. Myrna, "Automatic control of execution: An overview," *International Conference on APL (APL '79)*, 9(4): 141-147, 1979.
- [ABJ10] H. Abdul Basit and S. Jarzabek, "Towards structural clones - analysis and semi-automated detection of design-level similarities in software," *VDM*, 1-153, 2010.
- [BAK92] B. S. Baker, "A program for identifying duplicated code," *Computing Science and Statistics*, 24:49-57, 1992.
- [BAR10] L. Barbour, H. Yuan, and Y. Zou, "A technique for just-in-time clone detection in large scale systems," *International Conference on Program Comprehension (ICPC)*, 76-79, 2010.
- [BAS13] H. Bast and M. Celiklik, "Efficient fuzzy search in large text collections," *ACM Transactions on Information Systems*, 31(2), 2013.
- [BAX98] I. D. Baxter, A. Yahin, L. Moura, M. Sant'Anna, and L. Bier, "Clone detection using abstract syntax trees," *International Conference on Software Maintenance*, 368-377, 1998.
- [BEL07] S. Bellon, R. Koschke, G. Antoniol, J. Krinke, and E. Merlo, "Comparison and evaluation of clone detection tools," *IEEE Transactions on Software Engineering*, 33(9): 577-591, 2007.
- [BAZ11] S. Bazrafshan, R. Koschke, and N. Gode, "Approximate code search in program histories," *Working Conference on Reverse Engineering (WCRE)*, 109-118, 2011.
- [BRA10] J. Brandt, M. Dontcheva, M. Weskamp, S. R. Klemmer, "Example-centric programming: integrating web search into the development environment," *SIGCHI Conference on Human Factors in Computing Systems*, 513-522, 2010.
- [BR198] S. Brin, L. Page, "The anatomy of a large-scale hypertextual Web search engine," *Computer Networks and ISDN Systems*, 30(1):107-117, 1998.
- [CAR93] S. Carter, R. J. Frank, and D. S. W. Tansley, "Clone detection in telecommunications software systems: A neural net approach," *International Workshop on Applications of Neural Networks to Telecommunications*, 273-287, 1993.
- [CAU86] P. J. Caudill and A. Wirfs-Brock, "A third generation Smalltalk-80 implementation," *Conference on Object-oriented Programming Systems, Languages and Applications(OOPLSA '86)*, 21(11): 119-130, 1986.
- [DEE05] S. Deelstra, M. Sinnema, and J. Bosch, "Product derivation in software product families: a case study," *Journal of Systems and Software*, 74(2):173-194, 2005.
- [GJZ08] M. Gabel, L. Jiang, and Z. Su, "Scalable detection of semantic clones," *International Conference on Software Engineering (ICSE)*, 321-330, 2008.
- [GRI81] S. Grier, "A tool that detects plagiarism in Pascal programs," *SIGCSE Technical Symposium on Computer Science Education (SIGCSE '81)*, 13(1):15-20, 1981.
- [GRK09] N. Göde and R. Koschke, "Incremental clone detection," *European Conference on Software Maintenance and Reengineering (CSMR)*, 219-228, 2009.
- [HUM10] B. Hummel, E. Juergens, L. Heinemann, and M. Conradt, "Index-based code clone detection: incremental, distributed, scalable," *International Conference on Software Maintenance (ICSM)*, 1-9, 2010.
- [HUN77] J. W. Hunt and T. G. Szymanski, "A fast algorithm for computing longest common subsequences," *Communications of the ACM*, 20(5), 350-353, 1977.
- [JAC84] J. Jacobsen, "An automated management system for applications software," *ACM SIGUCCS Conference on User services (SIGUCCS '84)*, 173-175, 1984.
- [JAC01] P. Jaccard, "Étude comparative de la distribution florale dans une portion des Alpes et des Jura," *Bulletin de la Societe Vaudoise des Sciences Naturelles*, 37:547-579, 1901.
- [JAH10] P. Jablonski and D. Hou, "Aiding software maintenance with copy-and-paste clone-awareness," *International Conference on Program Comprehension (ICPC)*, 170-179, 2010.
- [JIA07] L. Jiang, G. Misherghi, Z. Su, and S. Glondu, "Deckard: Scalable and accurate tree-based detection of code clones," *International Conference on Software Engineering (ICSE '07)*, 96-105, 2007.
- [JMM09] Y. Jia, D. Binkley, M. Harman, J. Krinke and M. Matsushita, "KClone: a proposed approach to fast precise code clone detection," *International Workshop on Software Clones (IWSC)*, 2009.
- [KAM02] T. Kamiya, S. Kusumoto, and K. Inoue, "CCFinder: a multilinguistic token-based code clone detection system for large scale source code," *IEEE Transactions on Software Engineering*, 28(7): 654-670, 2002.
- [KAW09] S. Kawaguchi, T. Yamashina, H. Uwano, K. Fushida, Y. Kamei, M. Nagura, and H. Iida, "SHINOBI: A tool for automatic code clone detection in the IDE," *Working Conference on Reverse Engineering (WCRE)*, 313-314, 2009.
- [KEL83] M. I. Kellner, "Ten years of software maintenance: progress or promises?," *Conference on Software Maintenance (ICSM)*, 406-408, 1993.
- [KKI11] H. Kim, Y. Jung, S. Kim, K. Yi, "MeCC: memory comparison-based clone detector," *International Conference on Software Engineering (ICSE)*, 301-310, 2011.
- [KLX11] I. Keivanloo, J. Rilling, and P. Charland, "Internet-scale real-time code clone search via multi-level indexing," *Working Conference on Reverse Engineering (WCRE)*, 23-27, 2011.
- [KLZ11] I. Keivanloo, J. Rilling, and P. Charland, "SeClone-a hybrid approach to internet-scale real-time code clone search," *International Conference on Program Comprehension (ICPC)*, 223-224, 2011.
- [KLX12] I. Keivanloo, "Leveraging clone detection for Internet-scale source code search," *International Conference on Program Comprehension (ICPC)*, 277-280, 2012.
- [KLF12] I. Keivanloo, C. Forbes, A. Hmood, M. Erfani, C. Neal, G. Peristerakis, J. Rilling, "A Linked Data Platform for Mining Software Repositories," *Working Conference on Mining Software Repositories (MSR)*, 2012.
- [KON97] K. Kontogiannis, "Evaluation experiments on the detection of programming patterns using software metrics," *Working Conference on Reverse Engineering*, 44-54, 1997.
- [KOS12] R. Koschke, "Large-scale inter-system clone detection using suffix trees," *European Conference on Software Maintenance and Reengineering (CSMR)*, 309-318, 2012.
- [KNU77] D. E. Knuth, J. H. Morris, and V. R. Pratt, "Fast pattern matching in strings," *SIAM Journal on Computing* 6(2): 323-350, 1977.
- [KOS08] R. Koschke, "Frontiers of software clone management," *Frontiers of Software Maintenance (FoSM)*, 119-128, 2008.
- [LEM11] M. W. Lee, S. W. Hwang, and S. Kim, "Integrating code search into the development session," *International Conference on Data Engineering (ICDE)*, 1336-1339, 2011.
- [LER10] M. W. Lee, J. W. Roh, S. W. Hwang, and S. Kim, "Instant code clone search," *ACM SIGSOFT International Symposium on Foundations of Software Engineering (FSE'10)*, 167-176, 2010.
- [LOM83] J. V. Lombardi, "Computer literacy: The basic concepts and language," *Indiana University Press*, 1983.
- [MAN08] C. D. Manning, P. Raghavan, and H. Schütze, "Introduction to information retrieval," *Cambridge University Press*, 2008.
- [MAR01] A. Marcus and J. I. Maletic, "Identification of high-level concept clones in source code," *International Conference on Automated Software Engineering (ASE)*, 107-114, 2001.
- [MAY96] J. Mayrand, C. Leblanc, and E. M. Merlo, "Experiment on the automatic detection of function clones in a software system using metrics," *International Conference on Software Maintenance*, 244-253, 1996.
- [MIS04] G. Mishne and M. De Rijke, "Source code retrieval using conceptual similarity," *Conference on Computer Assisted Information Retrieval (RIA0'04)*, 539-554, 2004.
- [OTT76] K. J. Ottenstein, "An algorithmic approach to the detection and prevention of plagiarism," *ACM SIGCSE Bulletin*, 8(4): 30-41, 1976.
- [PER88] J. M. Perry, "Perspective on Software Reuse," *Technical Report, No. CMU/SEI-88-TR-22*, Carnegie-Mellon Univ. Pittsburgh PA Software Engineering

Inst, 1988.

[ROY09] C. K. Roy and J. R. Cordy, "A mutation/injection-based automatic framework for evaluating code clone detection tools," International Conference on Software Testing, Verification and Validation Workshops (ICSTW'09), 157-166, 2009.

[ROS09] C. K. Roy, J. R. Cordy, and R. Koschke, "Comparison and evaluation of code clone detection techniques and tools: A qualitative approach," Science of Computer Programming, 74(7): 470-495, 2009.

[SMI09] R. Smith and S. Horwitz, "Detecting and measuring similarity in code clones," International Workshop on Software Clones (IWSC), 2009.

[TAI09] R. Tairas and J. Gray, "An information retrieval process to aid in the analysis of code clones," Empirical Software Engineering, 14(1):33-56, 2009.

[TAN87] A. S. Tanenbaum, "A UNIX clone with source code for operating systems courses," ACM SIGOPS Operating Systems Review, 21(1):20-29, 1987.

[UCI10] C. Lopes, S. Bajracharya, J. Oshser, and P. Baldi, "UCI source code data sets," <http://www.ics.uci.edu/~lopes/datasets>, 2010.

[WAL03] A. Walenstein and A. Lakhota, "Clone detector evaluation can be improved: ideas from information retrieval," 2nd International Workshop on Detection of Software Clones (IWDSC), 2003.

[WEH03] H. J. Webber, "New horticultural and agricultural terms," Science, 18(459): 501-503, 1903.

[YHU11] Y. Higo and S. Kusumoto, "Code clone detection on specialized PDGs with heuristics," European Conference on Software Maintenance and Reengineering (CSMR), 75-84, 2011.

[ZIB12] M. F. Zibran and C. K. Roy, "IDE-based real-time focused search for near-miss clones," ACM Symposium on Applied Computing (SAC'12), 1235-1242, 2012.

APPENDIX - TRANSFORMATION FUNCTION DESIGN DISCUSSION

Several token types exist in source code such as method names, class names, primitive types, language keywords, variables, and constants. In general, apart from language keywords, which are consistent through the code, the token names can refer to different concepts. Despite differences in names, the semantic of tokens can still be similar (from algorithmic point of view). We refer to this case as tokens' semantic stability issue. Figure 11 provides an example where two code fragments can be considered clones even though they use different variable names (i.e., att and var).

```

...
5: String msg="exit 0";
6: for(AttributeEntity att : t.getAttributes())
7: {
...

...
5: String msg="exit 0";
6: for(AttributeEntity var : t.getAttributes())
7: {
...
    
```

Figure 11. Two code cloned code fragments that are using different variable names

It is a well-known practice (e.g., [KAM02]) in clone detection tools to replace all tokens with placeholders to reduce such syntactic and semantic dissimilarities. This practice is useful when the clone detection approach is not able to judge the semantics of the token based on its name and other available information (e.g., AST). In our

research, we proposed various transformation functions in order to be able to address different types of similarity. For example, the *c* function only preserves method names and class names. *c* replaces almost all other tokens with # as placeholder. We defined 5 transformation functions (Table 4) covering different scenarios and requirements. However, all of them preserve the method name tokens. For our approach, we decided to preserve method names, as we observed that method names have stable semantics in our research context (i.e., large-scale code search). Our observation is based on an analysis of a one-year query log of Koders [UCI10] (one of the state of the art code search engines). When analyzing the query log, we focused on 18 programming languages, which have a method construct as part of their language. This log contains a total of approximately 10 million records that we analyzed. As part of that analysis, we observed that for Internet-scale code search, method names play an essential role. Our analysis showed that if a method name was present as part of the query, code download occurred 98% of the time (Figure 12 - *MCQ values*), whereas the overall download rate is 69% (Figure 12 - *All values*). Note that in Web search activity mining, downloads/clicks on search results are interpreted as the result of a successful search. This observation shows the importance of method names in a code search and can be used as an indicator for method tokens' semantics stability from end-users' point of view. Therefore, all encoded code patterns generated by our transformation functions preserve the method names, which also provide the added benefit of reducing the number of false positive rates during the later matching.

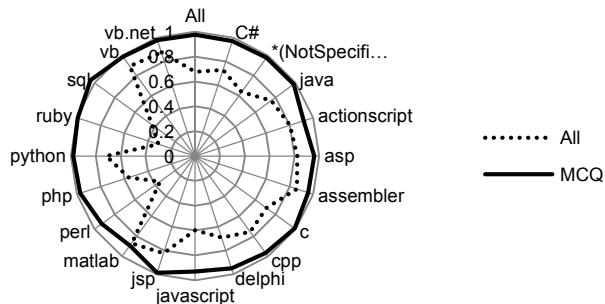


Figure 12. Importance of method names to the code search success rate – an indicator for method tokens' semantics stability from end-users' point of view.