Internet-scale Real-time Code Clone Search via Multi-level Indexing

Iman Keivanloo, Juergen Rilling
Dept. of Computer Science and Software Engineering
Concordia University
Montreal, Canada
{i_keiv, rilling@cse.concordia.ca}

Philippe Charland
System of Systems Section
Defence R&D Canada – Valcartier
Quebec, Canada
philippe.charland@drdc-rddc.gc.ca

Abstract–**Finding lines of code similar to a code fragment across large knowledge bases in fractions of a second is a new branch of code clone research also known as real-time code clone search. Among the requirements real-time code clone search has to meet are scalability, short response time, scalable incremental corpus updates, and support for type-1, type-2, and type-3 clones. We conducted a set of empirical studies on a large open source code corpus to gain insight about its characteristics. We used these results to design and optimize a multi-level indexing approach using hash table-based and binary search to improve Internet-scale real-time code clone search response time. Finally, we performed an evaluation on an Internet-scale corpus (1.5 million Java files and 266 MLOC). Our approach maintains a response time for 99.9% of clone searches in the microseconds range, while supporting the aforementioned requirements.**

Keywords–Code clone; real-time search; Internet-scale code search; code clone search; code clone detection

I. Introduction

Finding similar code fragments has been a major focus of code clone research and has resulted in various techniques. Although there exists a large body of research on code clone detection [1], the one on real-time clone search is limited. It is still a rather new research area, also known as just-in-time [2], real-time [3, 4], instant [5], or online clone search. It aims at finding all the fragments matching the input code fragment. Throughout this paper, we use the term *real-time* to emphasize the response time constraints the code clone search process has to satisfy.

Existing clone search approaches have traditionally focused on application domains such as software maintenance and plagiarism detection. In our research however, we are interested in the following open research question: Can clone search be successfully applied for Internet-scale code search [6] applications? We need a clone search approach that meets four major requirements: scalability, real-time response time, scalable incremental repository updates, and the support for several clone types (i.e., type-1, 2, and 3). The input is (1) a code fragment and (2) a target line which matches the relevant functionality in the result set from the structural query. In order to be able to take advantage of clone search during the source code search, the clone search granularity should be ideally at the line level to be able to match directly the target (input) line.

The main objective of the present research is to investigate if a clone search approach that consistently provides results in real-time (i.e., fractions of a second) can be derived, while supporting all aforementioned requirements. We applied a three-step research process to address our objective. First, we conducted a set of empirical studies to determine typical code patterns distribution across a large Internet-scale source code corpus. Our analysis included (1) unique patterns distribution (2) pattern frequencies, and (3) 32-bit hashing strength for code indexing using 1.5 million Java files. Second, we designed and implemented our clone search approach based on the first step result. Last, we evaluated its actual performance (response time). An overall observation from both our empirical studies and the performance evaluation of our multi-level fine-grained (e.g., single line) clone search approach is that it is indeed possible to achieve response times in the microseconds range over millions of files. For our analysis, we created our IJaDataset data set that contains 1,500,000 unique Java classes (~300 MLOC) from 18,000 Java open source projects [7]. To the best of our knowledge, this represents the largest *inter-project* Java source code data set (based on open source code) on which clone search has been applied so far. Related resources to this research are available online at http://aseg.cs.concordia.ca/seclone.

The remainder of this paper is organized as follows: Section 2 discusses basic terminology. Section 3 reviews related work. Statistical analysis results are presented in Sections 4 and 5. Our clone search approach, its computational complexity, and performance evaluation are reviewed in Sections 6-10.

II. Definitions and Basic Terms

A *fragment* is defined as a sequence (ordered) of source code lines. If two *code fragments* share some similarity, we call them a *clone pair*. In our research, we call the first participant the *querying fragment* and the second one, the *matching fragment*. Based on their actual similarity, clone pairs can be categorized [1] as *type-1*, *type-2*, *type-3*, and *type-4*. In addition to regular type-1, 2, and 3, there are some special clone types of particular interest which are defined in Table 1.

TABLE I. Special cases of clones

Querying Fragment	Clone Type	Matching Fragments
$\langle a, b, c\rangle^*$	Gapped (type3)	$\langle a, b, x, c\rangle$
	Mirrored	$\langle c, b, a\rangle$
	Unordered	$\langle c, a, b\rangle, \langle a, c, b\rangle, \langle b, a, c\rangle, \langle b, c, a\rangle$

* $\langle\,\rangle$ Denotes an *ordered sequence* of lines of code

III. Related Work

SHINOBI [8] introduced an online clone search approach using a suffix array index that is based on transformed tokens using CCFinder [9] transformation rules. A multidimensional token-level indexing approach has been introduced by Lee et al. [5] using an $R*\text{tree}$ on DECKARD's [10] approximate vector matching. Optimization on the repository size using sampling techniques is another approach to achieve scalable real-time clone search (e.g., Barbour et al. [2]). A more diverse approach to tree-based real-time clone search is hash table-based indexing. Hummel et al. [3] used a hash table to index 128-bit hash values of grouped lines of code. They were able to demonstrate that their approach reduces latency times compared to suffix tree-based search. Following the same indexing approach, we developed SeClone [4], which applies clustering on the clone pairs using the index-based search approach to separate false positives from true positives based on available metadata.

IV. Granularity Effect on Clone Search

The objective of this analysis was to determine if single-line or three-line (i.e., fine-grained) granularity is actually practical for real-time clone search over large amounts of data. Since the search granularity affects the overall clone search performance, we first had to determine the appropriate granularity for our clone search application. To answer this question, we performed a statistical analysis on our IJaDataset. We started our analysis by initially grouping source code fragments in chunks of three lines and then refined the analysis to a single-line granularity. Note that a Third Level Similarity (TLS) group denotes a set of potentially similar three-line code fragments (i.e., code clone). Similarly, First Level Similarity (FLS) is used for single-line patterns.

The first observation we made was that most TLS groups contain less than 2,000 members. There are only 1,220 patterns (outliers) out of 30,232,018 TLS which have more than 2,000 occurrences, representing less than 0.004% of all patterns in the corpus. Fig. 1 illustrates the distribution of TLS groups with less than 2,000 members. Based on these observations, it appears that the TLS heuristic tends to produce large numbers of small groups and very small numbers of large groups. On average, each TLS group has 2.37 members (Table 2). From this analysis, we can conclude that the TLS heuristic is practical for real-time clone search, as long as the outliers are handled properly, since (1) it distributes candidates in small-size groups and (2) for each query, only one group must be evaluated.

A. *What Does an Outlier Pattern Look Like?*

Table 3 shows the three largest groups with sample code. Based on our observations and earlier studies (e.g., [9]), these outliers are *insignificant* patterns from a clone search perspective and could be either excluded or controlled.

B. *FLS versus TLS Granularity*

In this section, we investigate the uniqueness of code lines to analyze and determine if it is practical to use single-line level granularity (i.e., FLS) instead of TLS. According to our data (Table 2), TLS distributes the candidates into 3.9 times more groups, while its group size is 6 times smaller than FLS's group size. Moreover, the outliers of FLS are much larger than TLS. Our analysis also shows that the TLS heuristic is considerably better than FLS for candidate selection phase given the trade-off between recall and response time. This observation constitutes the basis for our multi-level indexing clone search where we are going to first find candidates based on 3-line similarity (first level index) and then apply coverage discovery using the single-line similarity (second level index).

V. Distinct Pattern per File Analysis

One research question we addressed in an early stage of this research was if it is possible to identify a subset of files that is representative for an entire data set while supporting a similar pattern/clone recall at the fragment level. Fig. 2 shows our observation of how many new patterns were introduced in the index when we selected random subsets, each of them consisting of 50K files. For this study, we considered the introduction of all patterns, even unique ones. Fig. 2 shows that for the more popular patterns, it is indeed possible to use a subset as data input. For example, 33% of the data contains 91% of popular patterns (15 occurrences or more) and 50% of the data set includes 71% of patterns with at least two occurrences. Based on this observation, we can consider random selection in our research context as an acceptable approach to reduce the data set size.

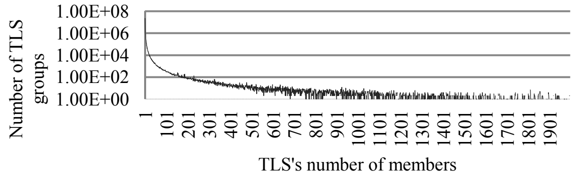


Figure 1. TLS group size (i.e., 3-line pattern occurence frequency) distribution.

TABLE II. TLS and FLS Charactrestics

Property	Value	
	TLS	*FLS*
Number of groups	30,232,018	7,606,433
Number of members	71,911,376	71,911,376
Number of single-member groups	22,824,697	4,770,010
Largest group size (occurrence)	1,048,575	2,937,700
Average (group size)	2.37	9.45
Standard deviation (group size)	293.23	1898.75

TABLE III. Outlier code patterns

Rank	Number of Occurrences	Pattern Title	Sample Code
1	1304840	Local getter	`method() { return variable; }`
2	636846	General setter	`method(type arg) { this.variable = arg; }`
3	246082	General getter	`method() { return variabale.property; }`

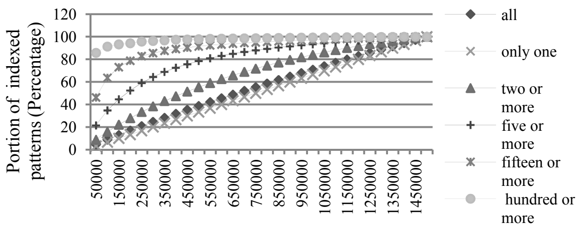


Figure 2. Analysis of new 3-line code patterns per file introduced in the index. Patterns are categorized based on their total # of occurrences.

VI. Scalable Real-Time Code Clone Search

Based on the results from the previous sections, we designed a clone search engine, which is based on our earlier research architecture (SeClone [4]).

Preprocessing. For the line-based search approach, code layout unification (i.e., formatting) is an essential step to increase recall [9]. The preprocessing step uses the ASTs (Abstract Syntax Tree) to parse the repository file by file and create a uniform fact representation. At the same time, the transformation process is triggered [3, 9], which includes one new rule. We added this rule after analyzing a one-year query log of Koders [7]. When analyzing the query log (10 million records), we focused on 18 programming languages. 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 in a query, code download occurred in 98% of the time. Without the method name, the download rate was only 69%. Since method names represent such a significant information source, we decided to add a special transformation rule to preserve method names.

We use hash codes instead of lines of code to decrease the main memory consumption, while reducing search latency times in the order of several magnitudes. Hash codes were created at two granularity levels: TLS and FLS. FLS refers to hash codes for each single transformed line of code, whereas TLS groups are sequences of three lines of code, which are assigned a hash code. The TLS heuristic was introduced to improve performance, by identifying matching code candidates for a run-time query (based on studies reported in previous sections). FLS is complementary to the TLS heuristic, as it provides the ability of a more fine-grained clone coverage discovery. A simplified example is shown in Fig. 3.

Indexing. We deploy a multi-level indexing approach to achieve fast response times and at the same, be able to support type-1, type-2, and type-3 code clone search. We create three types of indices. There exists only one TLS index, which is a hash table with each cell of the index containing all similar 3-line fragments. There are two types of indices for each file, a primary and a secondary FLS index. The primary FLS provides sequential access to source code lines. The secondary FLS supports random access. All secondary index cells are linked to their corresponding cells in the primary index to allow for $O(1)$ round-trip complexity. For our approach to support type-3 code clone search, it requires the use of the following two additional steps. The first one resides within the TLS hash code generation. After calculating the FLS hash codes, members of each *TLS entity* are sorted (Fig. 3) to remove ordering from the TLS hash codes. It is useful to detect *mirrored* and *unordered* code clone candidates. The other step is the use of the secondary FLS index to locate a line of code via its FLS hash code. This allows us to improve the performance for *gapped* (type-3) *clone* to logarithmic complexity, versus the linear complexity of the primary FLS index for the type-3 code clone search.

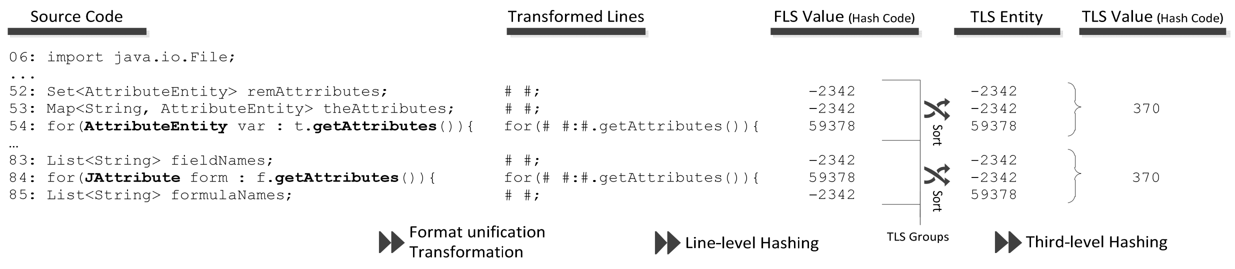
Searching. The search algorithm (Fig. 4) accepts a code fragment and a target line number within this fragment known as the *querying fragment*. In general, it finds all files with lines similar to the TLS at the target line. Our approach will try to detect the maximum coverage for each pair using a line by line comparison, starting from the target line, using the FLS indices. For the type-3 clone pair search, the algorithm searches until it reaches a point where two lines are no longer equal. In this case, the algorithm will locate the closest similar line using the secondary FLS index.

A. *Is a 32-bit Hash Code Strong Enough for Clone Search?*

In order to reduce the memory and processing consumption, we used 32-bit hash codes instead of the 128-bit hashing used in [3]. In theory, 32-bit hash code-based indexing is not as strong as 128-bit hashing. Duplication in assigned hash keys occurs in two sections which are TLSs and FLSs. We addressed the first type of inaccuracies by applying a verification step that compares TLS and FLS values during candidate search. Since we are unable to detect inaccuracies in FLSs' hash keys, we measured its inaccuracy in code clone search. For the analysis of the practicality and applicability of 32-bit hash keys for code clone search, we conducted a comprehensive study on more than one million Java files (IJaDataset). We created 32-bit hash keys for all transformed lines of code (i.e., FLSs) using the JDK standard hash function for String. We also analyzed the error rate when one key is used for more than two distinct lines. Surprisingly, the results showed us that 32-bit hash keys are strong enough (i.e., 0.002% error rate) to be used for indexing source code for clone search purpose.

VII. Computational complexity

In this section, we discuss the computational complexity of our clone search approach, which is summarized in Table 4. T represents the TLS index size which is $O(n)$, with n being the size of the corpus in terms of LOC. p_t denotes the TLS group (i.e., t) size. m denotes the total number of detected clone pairs, the average size of clone pairs is represented by c, and the total number of updated lines of code by l. In general, TLS index expected lookup is $O(1)$, since it is hash table-based. Sequential access to the primary FLS index and random access to the secondary FLS are $O(1)$ and $O(\log r)$ respectively, where r denotes file size (LOC). Logarithmic order is due to the underlying binary search on sorted FLS indices. Note that $O(\log r)$ is negligible for regular files (e.g., 1K LOC).



```
06: import java.io.File;
...
52: Set<AttributeEntity> remAttrributes;            # #;                               -2342        -2342
53: Map<String, AttributeEntity> theAttributes;     # #;                               -2342        -2342      370
54: for(AttributeEntity var : t.getAttributes()){   for(# #:#.getAttributes()){        59378        59378
…
83: List<String> fieldNames;                        # #;                               -2342        -2342
84: for(JAttribute form : f.getAttributes()){       for(# #:#.getAttributes()){        59378        -2342      370
85: List<String> formulaNames;                      # #;                               -2342        59378
```

Figure 3. Transformation, FLS, TLS grouping, and hashing samples

Based on our analysis, the following observations can be made. First, the preparation step complexity is linear, which is important since our approach will have to support Internet-scale code analysis. Second, the clone search is almost constant, since $c, r \; and \; p_t \ll n$. Third, the update complexity is linear to the number of updated lines. As a result, our approach has *the lowest observed complexity* for clone search (excluding type-3 clones), repository preparation, and updates. Both the complexity and the memory requirements support our claims that it can support scalable incremental repository updates and provide real-time clone search for very large source code corpora.

VIII. Clone pair Detection Experiment

Although the main objective of our research is clone *search*, we conducted a traditional clone *detection* experiment to determine if our clone search approach can also be applied for clone detection on an Internet-scale corpus to detect type-1, 2, and 3 clones. We applied the detection on our complete repository (1,500,000 files) and the results are shown in Fig. 5. In our experiment, the worst-case scenario took about 21 minutes to find all possible raw clone pairs (11 billion).

Our results provide a clear indication of the number of clone pairs that an Internet-scale clone detection tool must be able to handle in order to be scalable. In our case, the run-time would remain under 3 minutes for detection of all clone-pairs (within the entire repository) if we exclude the outlier patterns.

TABLE IV. Computational Complexity

Process	Time	Memory
Repository preparation (indexing)	$O(n)$	$\boldsymbol{O(n+T)}$
Clone pair detection	$O(\sum_{t \epsilon T} p_t^2)$	$\boldsymbol{O(m)}$
Clone pair search	$O(p_t * c \log r)$[a]	$\boldsymbol{O(p_t)}$
Repository update (addition/deletion)	$O(l)$	$\boldsymbol{O(l+T_l)}$

a. $\log r$ is applicable in case of access to the secondary FLS index. Similar situations can be applied to the forward detection complexity.

```
Algorithm findSimilarPairs($f, n, ix_{tls}$)
Input   $f$: query's code fragment, $n$:target line in $f$, $ix$: indices
Output  $CP, CP_{err}$: true positive and false positive clone-pair sets

1  $ix_{fls} \leftarrow calculateFLS(f)$        // $ix_{fls}$: FLS values for $f$
2  $TLS[] \leftarrow ix_{tls}.find(calculateTLS(ix_{fls}, n))$
3  for  $tls$  in  $TLS$                        //$tls$: minimal candidate lines
4          $ix'_{fls} \leftarrow retrieve\ primary\ fls\ index\ of\ tls$
5          $n' \leftarrow retrieve\ target\ line\ of\ tls$
           //$f'$ is the candidate fragment
6          if $\{f'_{fls}[n'], f'_{fls}[n'_{prev}], f'_{fls}[n'_{next}]\}$
                 $-\{f_{fls}[n], f_{fls}[n_{prev}], f_{fls}[n_{next}]\} = \emptyset$  then
7                  $cp.up \leftarrow findPairCoverage(ix_{fls}, n, ix'_{fls}, n', up)$
8                  $cp.dn \leftarrow findPairCoverage(ix_{fls}, n, ix'_{fls}, n', dwn)$
9                  $CP.add(cp)$
10         else
11                 $CP_{err}.add(tls)$ //false positive due to hashing
-----------------------------------------------------------------------------
Algorithm findPairCoverage($ix_{fls}, n, ix'_{fls}, n', d$)
Input   $d$:lookup direction. $n$ and $n'$:pointers to the lines.
Output  $cp$'s coverage boundary for the given direction (i.e. $d$)
/* $t$ denotes the distance threshold for type-3 approximate lookup. The default
value is set to the current method block size */

1  $cp.type = general$     //$cp$ is the clone-pair under evaluation
2  if      $n$'s $tls$ is not equal to $n'$'s $tls$ considering the line order then
3          $cp.type = type\text{-}3$
4  do      // do-while loop
//$n_d$ and $n'_d$ denote the $\frac{next}{prev}$ lines of $n$ and $n'$ regarding $d$
5          if  $ix_{fls}[n_d] = ix'_{fls}[n'_d]$   then
6                  $n \leftarrow n_d; n' \leftarrow n'_d$
7                  continue loop
8          else
9                  $ix'_{fls'} \leftarrow retrieve\ secondry\ index\ of\ ix'_{fls}$
10                 $M \leftarrow ix'_{fls'}.find(ix_{fls}[n_d])$
11                 $x \leftarrow m: m \in M - cp, \forall m' \in M - cp$
                                   $, |n' - m| < |n' - m'|$
12                 if  $|n'_d - x| < t$  then
13                         $n \leftarrow n_d; n' \leftarrow x$ ; $cp.type = type\text{-}3$
14                         continue loop
15                 else
16                         fix corresponding boundary of $cp$
17                         break loop
18 while($true$)   //end of do-while loop
```

Figure 4. Clone pair matching and coverage search algorithms

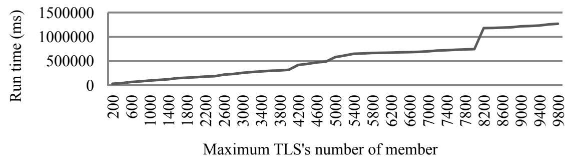


Figure 5. Clone pair detection run-time. Note that it is different from the clone search process.

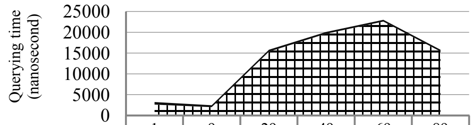
IX. CLONE SEARCH PERFORMANCE EVALUATION

So far, we have shown that a real-time clone search approach for Internet-scale code search should be feasible. As part of our evaluation, we implemented and conducted a set of performance evaluations using our clone search approach to confirm our earlier observations. As discussed earlier, our approach was designed from the beginning as an Internet-scale real-time code clone search with response time being the most critical requirement such a clone search approach has to meet. Although precision and recall are important, they play in this context only a secondary role. For our approach, we observed similar precision and recall as one reported by CCFinder [9], which was expected due to the similarities of the transformation techniques.

Fig. 6 shows the response time when answering queries related to regular patterns (from 1 to 90 occurrences). Note that the times are presented in the nanosecond (ns) and microsecond (μs) scale. The x-axis represents the number of detected clone pairs. A typical query will be answered in less than $25\mu s$. We also conducted additional experiments for queries with a larger number of results being returned, by including queries covering popular and outlier patterns (Fig. 7). As shown, it is possible to answer almost all queries (queries related to 99.99% of code patterns) in less than 900 microseconds using our multi-level indexing approach. For the outliers (i.e., 0.004% of all patterns), the response time increases to 1 second (worst-case observed).

X. COMPARISSION WITH OTHER INDEX-BASED APPROACHES

Excluding our general research contributions (e.g., studying the effect of pattern frequencies on search response times), our clone search approach advances the state-of-the-art in index-based clone search research. First, our multi-level indexing approach lets us support type-3 clones (including the other especial clone types), while the others just support type-2 clones. Second, it was believed that 32-bit hash values are not strong enough for a clone search approach. In this research, we showed that they are 99.9% reliable. Third, we improved the overall performance, by using about 75% less memory for keeping hash values. Moreover, based on our observations, our approach outperforms Hummel et al. clone search [3] by a factor of 2 to 9, depending on the search context. Furthermore, compared to the other approaches discussed in the related work section, which have response times in the milliseconds range, the response times of our approach are on average in the microseconds range.



	1	9	29	49	69	89
■ TLS Lookup Time	607	452	424	410	433	430
+ Clone-pair Lookup	2639	2032	15381	19665	22578	15392

Figure 6. Clone pair search response time for regular patterns

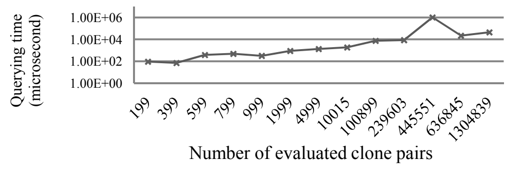


Figure 7. Querying time for common (199 - 2K) and outliers (2K - 1M) pattern. Our approach asnwers 99.99% of patterns in less than 900 μs.

XI. CONCLUSION

In this research, we investigated if a clone search approach can be derived for Internet-scale code search. The objective was to provide an approach that consistently provides results in real-time (fractions of a second), while being scalable to large corpora (millions of files). We applied a three-step research process to answer our questions. First, we conducted a set of empirical studies to determine typical code patterns occurrence frequency across a large Internet scale source code corpus. Our analysis included (1) pattern distributions at the file level (2) pattern frequencies and (3) 32-bit hashing strength code indexing. In the second step, we used the insights from this analysis to design and implement a multi-level index-based clone search approach. In the last step, we evaluated the actual performance (response time) of our approach. In the empirical studies of our multi-level fine-grained (e.g., single line) clone search approach, we observed that it is possible to achieve response times in the microseconds range when searching for clones over millions of files. As part of future work, we plan to adopt index compression and improve the recall of type-3 clones.

REFERENCES

[1] 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*, vol. 74, May. 2009, pp. 470-495.

[2] L. Barbour, H. Yuan, and Y. Zou, “A technique for just-in-time clone detection in large scale systems,” *Proc. 18th IEEE International Conf. on Program Comprehension*, 2010.

[3] B. Hummel, E. Juergens, L. Heinemann, and M. Conradt, “Index-based code clone detection: incremental, distributed, scalable,” *Proc. 32th IEEE International Conference on Software Engineering*, 2010.

[4] I. Keivanloo, J. Rilling, and P. Charland, “SeClone - A hybrid approach to internet-scale real-time code clone search,” *Proc. 19th IEEE Intl. Conf. on Program Comprehension, Tool Demo*, 2011.

[5] M.W. Lee, .J.W. Roh, S.W. Hwang, and S. Kim, “Instant code clone search,” *Proc. 18th ACM SIGSOFT International Symp. on Foundations of Software Engineering*, 2010.

[6] I. Keivanloo, L. Roostapour, P. Schugerl, and J. Rilling, “Semantic Web-based Source Code Search,” *Proc. 6th Intl. Workshop on Semantic Web Enabled Software Engineering (SWESE),* 2010.

[7] C. Lopes, S. Bajracharya, J. Ossher, P. Baldi. UCI Source Code Data Sets. 2010. http://www.ics.uci.edu/~lopes/datasets.

[8] 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,” *Proc. 16th Conf. Reverse Engineering*, 2009.

[9] T. Kamiya, S. Kusumoto, and K. Inoue, “CCFinder : a multilinguistic token-Based code clone detection system for large scale source code,” *IEEE Trans. Software Engineering,* vol. 28, 2002, pp. 654-670.

[10] L. Jiang, G. Misherghi, Z. Su, and S. Glondu, “Deckard: Scalable and accurate tree-based detection of code clones,”*, Proc. 29th IEEE International Conf. on Software Engineering,* 2007.