DCCD: An Efficient and Scalable Distributed Code Clone Detection Technique for Big Code

Junaid Akram*(Member, IEEE), Zhendong Shi*, Majid Mumtaz* and Luo Ping*
*State Key Laboratory of Information Security, School of Software Engineering, Tsinghua University China.
Email: [znd15, szd15, maji16]@mails.tsinghua.edu.cn
Email: luop@mail.tsinghua.edu.cn

***Abstract*—Code clone detection is a very hot topic in the field of software maintenance, reuseability and security. There is still a lack of techniques to detect near-miss clones at different level of granularities, especially in big code. This paper presents Distributed Code Clone Detection (DCCD) technique, which detects clones from big code bases based on feature extraction. We performed preprocessing, indexing and clone detection for almost 27 TB of source code (324 billion LOC), DCCD is quite faster and efficient as compared to existing distributed indexing and clone detection techniques, i.e. 36 times faster than Benjamin technique, which is 86 times faster than CCFinder. These two techniques are also distributed and just detect Type-1 and Type-2 clones, but our technique DCCD even detects Type-3 clones, efficiently. Our approach is faster, flexible, scalable and provides 87% accurate results with authenticity, ease of accessibility, upgradeability and maintainability.**

keyword Clone detection, Software maintenance, Software reuse, Big code, Similarity/Plagiarism detection

I. Introduction

When a programmer copies code fragments and tries to reuse them by pasting in other code sections with or without making minor modifications, this type of code reuse approach is called code cloning, and the pasted code fragment called a clone of the original. It is a very adapting process in software development activities. However, during detection of clones, it is hard to say that which code fragment is original and which code fragment is copied. Code clones bring troubles in software security and maintenance and they lead to bug propagation. Roy describes that a very significant range (7% - 23%) of code is cloned in large scale systems [1]. Code clone detection techniques are very helpful for code maintainability, code plagiarism detection [2] [3], code verification, copyright detection, security flaws detection, detection of bugs and malicious software detection.

Our developed platform provides an index-based hybrid solution (semantic approach) by combining different clone detection techniques for large scale systems which is distributed, scalable and incrementable. It detects Type-1, Type-2, Type-3 code clones in real time environment on the basis of big code. Our system is based on Hadoop environment, which extends a practical applicability of index based clone detection for very large code bases and it demonstrates the response time sufficiently fast. Our technique has been developed and tested to detect code clones in 15 different programming languages i.e. Java, JavaScript, C, C++, C#, Xml, Python, Php, Sql, Vb, Cobol, Text, Ruby, Ada, Matlab. In this paper, we will discuss and display results of 3 programming languages on large scale level, which are Java, JavaScript and C/C++. The downloaded source code was about 40 TB (20 TB C & C++, 10 TB Java, 10 TB JavaScript), but the experiment of preprocessing, indexing and feature extraction was performed on almost 27 TB (C & C++: 16 TB, JavaScript: 11 TB, Java: 1 TB) source code. The main purpose of our research was to deal with the big code on a large-scale level and detect near-miss clones accurately, which is not only challenging but computationally expensive. We preprocess source code using different mining techniques/filters, extracting main features and store the index information into a database for further inspection process of clone detection. Preprocessing, normalization, feature extraction and indexing process were performed in a pipeline, so index creation is actually fully depended on the output of preprocessing of source code. The top level view of whole system has been shown in Fig 1. Index based [4] and CCFinder [5] are two token-based distributed clone detection techniques but they just only detect Type-1 & Type-2 clones, and even not support big code indexing and detection process. Our approach detects Type-1, Type-2, Type-3 clones with high accuracy rate, meanwhile it's incremental, scalable and fast.

II. DCCD Architecture

In this section, we briefly describe all the steps and phases of the proposed DCCD clone detection architecture. The proposed work is the hybrid technique of clone detection for large-scale systems, in which we have applied many screening filters to retrieve exact clone files from big code bases. Clone detection process comprises of many phases, where each next phase depends on the previous phase and builds on the outcome of the previous phase. There are four main phases in our clone detection approach as shown in Fig 1. Fig 2 shows the technical view of preprocessing and normalization process. The left side of Fig 2, correspondence between an original code fragment and the preprocessed and normalized code fragment is visualized. On the right side of Fig 2, the indexing entities and chunk properties have been shown, where MD5 are the hash values of the prefix and suffix sequence of statements, of the source code file. The reason we used a sequence of statement during indexing instead of individual statement is that the sequence of statements are more unique and identical. FNV values are the 10 hash values per division

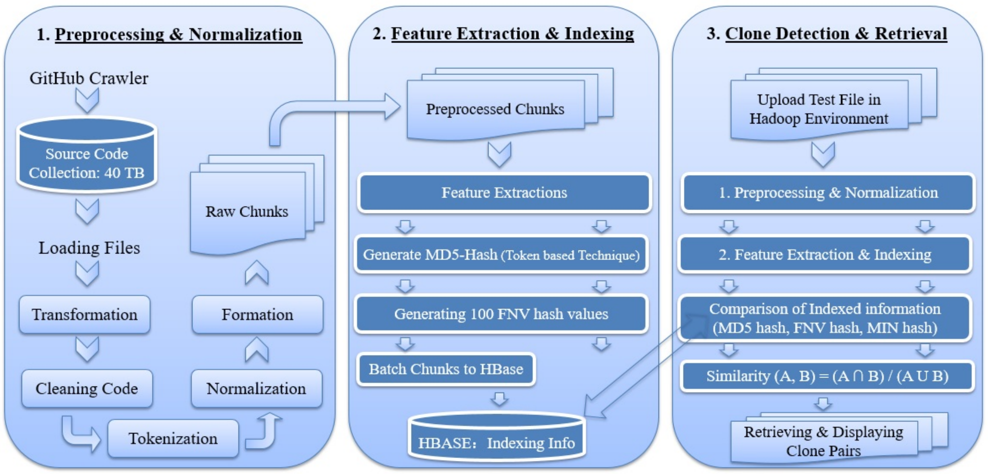
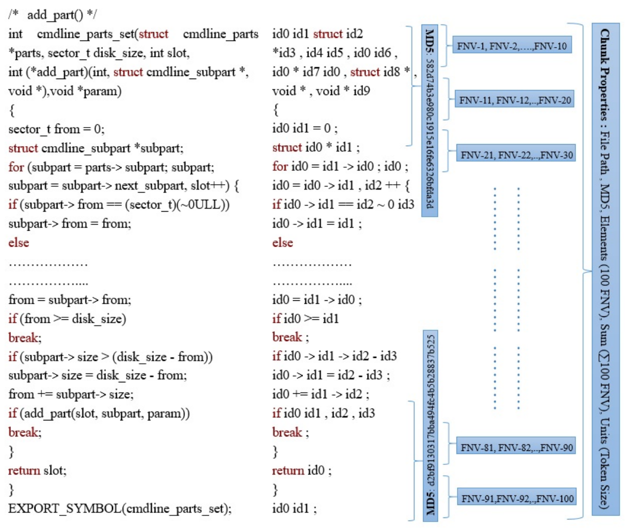

1. Preprocessing & Normalization
GitHub Crawler
Source Code Collection: 40 TB
Loading Files
Transformation
Cleaning Code
Tokenization
Normalization
Formation
Raw Chunks
2. Feature Extraction & Indexing
Preprocessed Chunks
Feature Extractions
Generate MD5-Hash (Token based Technique)
Generating 100 FNV hash values
Batch Chunks to HBase
HBASE: Indexing Info
3. Clone Detection & Retrieval
Upload Test File in Hadoop Environment
1. Preprocessing & Normalization
2. Feature Extraction & Indexing
Comparison of Indexed information (MD5 hash, FNV hash, MIN hash)
Similarity (A, B) = (A ∩ B) / (A U B)
Retrieving & Displaying Clone Pairs

Fig. 1: DCCD System top level view



```
/*  add_part() */
int   cmdline_parts_set(struct   cmdline_parts
*parts, sector_t disk_size, int slot,
int (*add_part)(int, struct cmdline_subpart *,
void *),void *param)
{
sector_t from = 0;
struct cmdline_subpart *subpart;
for (subpart = parts-> subpart; subpart;
subpart = subpart-> next_subpart, slot++) {
if (subpart-> from == (sector_t)(~0ULL))
subpart-> from = from;
else
………………
……………....
from = subpart-> from;
if (from >= disk_size)
break;
if (subpart-> size > (disk_size - from))
subpart-> size = disk_size - from;
from += subpart-> size;
if (add_part(slot, subpart, param))
break;
}
return slot;
}
EXPORT_SYMBOL(cmdline_parts_set);
```

```
id0 id1 struct id2
*id3 , id4 id5 , id0 id6 ,
id0 * id7 id0 , struct id8 * ,
void * , void * id9
{
id0 id1 = 0 ;
struct id0 * id1 ;
for id0 = id1 -> id0 ; id0 ;
id0 = id0 -> id1 , id2 ++ {
if id0 -> id1 == id2 ~ 0 id3
id0 -> id1 = id1 ;
else
………………
……………....
id0 = id1 -> id0 ;
if id0 >= id1
break ;
if id0 -> id1 -> id2 - id3
id0 -> id1 = id2 - id3 ;
id0 += id1 -> id2 ;
if id0 id1 , id2 , id3
break ;
}
return id0 ;
}
id0 id1 ;
```

MD5: 582d74b3e980c1915e16fe6326bfda3d
MD5: d2bf9130317bba494fc4b5b28837b525
FNV-1, FNV-2,….,FNV-10
FNV-11, FNV-12,..,FNV-20
FNV-21, FNV-22,..,FNV-30
FNV-81, FNV-82,..,FNV-90
FNV-91,FNV-92,.,FNV-100
Chunk Properties : File Path , MD5, Elements (100 FNV), Sum (∑100 FNV), Units (Token Size)

Fig. 2: The original source code (left), its normalization (middle) and indexing info (right)

of a file. The Chunk properties are the feature, which we use for comparison of two files.

A. DCCD Preprocessing and Normalization

This is the first phase, which removes uninteresting and unwanted pieces of codes from source files. There are three major processes of this phase. (a) It reads source code files from disk and splits the code into tokens. (b) Remove all uninteresting pieces of code from the source code files. (c) Normalization is performed on these tokens. These tasks have been explained below in detail.

Loading: It reads the selected project files from the hard disk and load into RAM for further processing.

Transformation: In this task, we select the language type of project and consider the related source code files of that project and ignore all unwanted files.

Tokenization: After full transformation of source code into a byte stream, we start tokenizing the code sequence. The main purpose of this process is to organize source code in tokenized form line by line. Then lexical analyzer uses each traversal at the same time to tokenize. Finally, the corresponding token sequences are generated and stored in memory.

Normalization: During normalization, we delete the unwanted tokens, comments, spaces, import libraries and the tokens which do not have any effect on the source code.

Chunk Formation: After token normalization, we initialize an empty entity set (chunk), each attribute in the entity set initialized as null. We use this formation to make chunk ready to store Elements (feature), meanwhile, we identify the size of the chunk. After performing this task, we able to have preprocessed code, in which identifiers replaced with ID plus numbers (starts from 0); a string replaced by an empty string; fixed string with some identified characters; floating types with 0, and boolean values into true as shown in Fig 2.

B. Feature Extraction

This phase is the core part of our clone detection technique because these extracted features help us to detect code clones. Feature extraction basically involves reducing the amount of code to describe a big code base by extracting features from it. This phase consists of many steps, some of them performed in parallel and some of them performed in pipeline. The left side of Fig 3 shows the flow of each process into 5 steps, the right side displays the HBase entities, which we extract and store into HBase, i.e. `Row_key`, `Origin_id`, Elements, Units and All.

Step 1: This step actually gets the source code project files from code repository in pipeline. The source code of these files further converted into bytecode by using preprocessing and normalization phase.

Step 2: In this step, we use MD5 hashing algorithm as a feature extraction from the source file to differentiate the uniqueness of every file. MD5 encrypt the prefix 15 token statements and suffix 15 token statements of every source code

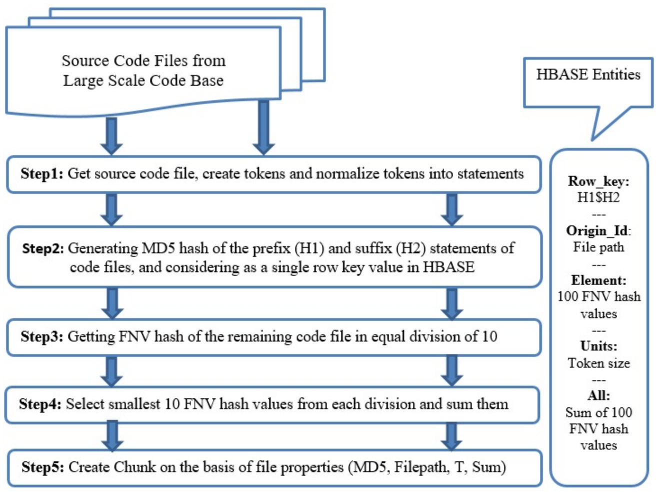

Source Code Files from Large Scale Code Base
HBASE Entities
Step1: Get source code file, create tokens and normalize tokens into statements
Step2: Generating MD5 hash of the prefix (H1) and suffix (H2) statements of code files, and considering as a single row key value in HBASE
Step3: Getting FNV hash of the remaining code file in equal division of 10
Step4: Select smallest 10 FNV hash values from each division and sum them
Step5: Create Chunk on the basis of file properties (MD5, Filepath, T, Sum)
Row_key: H1$H2
Origin_Id: File path
Element: 100 FNV hash values
Units: Token size
All: Sum of 100 FNV hash values

Fig. 3: Feature extraction from source code files

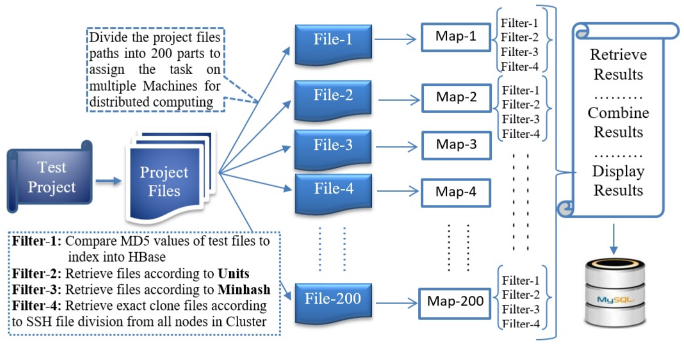

Divide the project files paths into 200 parts to assign the task on multiple Machines for distributed computing
Test Project
Project Files
File-1
File-2
File-3
File-4
File-200
Map-1
Map-2
Map-3
Map-4
Map-200
Filter-1
Filter-2
Filter-3
Filter-4
Retrieve Results
Combine Results
Display Results
Filter-1: Compare MD5 values of test files to index into HBase
Filter-2: Retrieve files according to Units
Filter-3: Retrieve files according to Minhash
Filter-4: Retrieve exact clone files according to SSH file division from all nodes in Cluster

Fig. 4: MapReduce view for clone retrievals

file. The reason to use 15 tokens of prefix and suffix is to define a unique signature set of every source file.

Step 3: In this step, we extract another feature from the source code files by using FNV (Fowler-Noll-Vo)[1] hashing algorithm. The reason we use FNV is that it quickly hashes the large amounts of data with a small conflict. The FNV hash has been generated for every token of the source file.

Step 4: In this case, we collect 100 hash values in total then add all of these values into one single value (long integer). These FNV hash values actually represent the overall characteristics of a file.

Step 5: The final step of feature extraction is to creating chunks on the basis of extracted features in previous steps.

C. Feature-Based Index Creation

Indexing allow to find all clones against a single file or for an entire system. Meanwhile it allows to update index info, when files are removed, modified, or added. Indexing based on feature extraction is the main data structure in our code clone detection technique. The index creation process is very flexible, fast, accurate, easy to maintain and upgradeable. We can update, delete or edit index information from HBase of any file at any time. Meanwhile, we can keep track and retrieve the index information of any file in less than a microsecond. The indexing data consists of following a list of labels, which describes the entities of HBase.

`Row_key`: It is MD5 hash value of the prefix (H1) and suffix (H2) statements of the source code files.

`Origin_id`: It is the file path or location of a file.

Element: It is 100 FNV hash values of a file.

Units: It is the total size of tokens inside the concerned file.

All: It is the sum of 100 FNV hash values.

To build an index of big code, we used Hadoop distributed framework. There are 7 systems in the cluster, one Master (Intel i7, 32GB) and six Slaves (Intel i5, 16GB). To perform experiment, we built an index of almost 27 TB of source code (16 TB C/C++, 10 TB JavaScript and 1TB of Java). This collection of source code for indexing was consisting of 1,039,260 projects, 885 million files and 324 billion of LOC.

[1]http://www.isthe.com/chongo/tech/comp/fnv/index.html

D. Code Clone Detection and Retrieval

This is the final phase, in which we retrieve and display the similarity between systems at different level of granularities, i.e. method level, chunk level, file level and project level. During detection, we filter, extract and retrieve all cloning objects, which meets the defined cloning filter conditions (Filter-1,2,3,4) as shown in Fig 4. The mapper (map reduce) retrieves all code clones and calculates the fraction of all files in the index, which at least contained one clone. We extract all clones against every single file of the test project.

In the first filter, the detector detects all the cloning files from big code repository by comparing indexes, which have the same `Row_key` (M1 hash $ M2 hash) values, where M1 is the hash value of prefix token statements and M2 is the hash value of suffix token statements. Then, we apply another filter to get more abstract cloning files from large scale code. In this process, we compare the sum of 100 FNV hash values (All) in the index database, which were stored in HBase during the building of index of big code. We continue applying filters to get abstract scale code base from big code. In next filter, we compare the 100 FNV hash values (Elements) of the test file in the index and retrieve the resulted clone files. This Element entity of index consists of the 100 FNV hash values. Finally, after getting the abstract cloning files, we further evaluate them for *exact clones* by using Minhash algorithm. According to the idea of Minhash algorithm, the similarity can be calculated by the following formula. Similarity (A, B) = ($A \cap B$ / A U B). Where $A \cap B$ is the number of code fragments, which are same in file A and in file B, A U B represents the total number of different code fragments in file A and in file B. In last step of detection and retrieval of code clone files. The detector uses SSH (Secure Shell) protocol to send, *to be tested files* in distributed environment to every attached node in the cluster. These files divided into chunks, generating the md5 hash values of defined chunks size (adjustable) and perform the further comparisons to detect exact clones. After getting same code fragments the detector evaluates the results, combined them and display to the users.

TABLE I: Building Index Results

Language	Size	Time (hours)	No of Projects	No of Files (millions)	LOC (billions)	Query Response per file (sec)
C & C++	16 TB	135	548,150	493	207	1.30
Java	1 TB	6	17,822	14	4	0.75
JavaScript	10 TB	65	473,228	378	113	0.95

Configurable Threshold Value: The threshold value is fully configurable by the user. It is also possible to detect clone fragments at different level of granularities, i.e. chunk level, method level, file level and project level. We note that we found the precision and recall to be optimum at 70% threshold. If we set threshold value high than 70%, the retrieved results will be more of Type-3, but it will be very less effect on Type-1 and Type-2 clones.

III. CASE STUDIES AND RESULTS

In this section, we summarize the implementation and results of our proposed technique (DCCD) for batch clone detection and distributed clone detection in big code. Here are different case studies and RQs (research questions) on collection of big code, indexing, response time, detection of clones and their results comparison. Here are the main RQs, we formulated in our study:

- RQ1: How is the speed of index creation and how much storage space does it occupy in memory?
- RQ2: In what semantics DCCD produce better results as compare to previous state-of-the-art methods?
- RQ3: Does DCCD support partial level and full level similarities detection?
- RQ4: Is the DCCD technique scalable and efficient as compared to other clone detection techniques?

A. Collection of Source Code

To perform the assessment test on DCCD, the big code was required for our source repository. We collected almost 40 TB (C & C++: 20TB, Java: 10 TB, JavaScript: 10 TB) of source code. Our big code collection was consisting on thousands of projects, millions of project files and billions of LOC.

B. RQ1: Indexing Speed and Storage

Hadoop distributed environment is used to build an index of large scale system in a fast and efficient way. There were 7 machines (1: Intel i7, 32GB and 6: Intel i5, 16GB) in Hadoop environment. But only 5 Machines used for performing pre-processing and building index simultaneously. Table I shows the indexing results of 27 TB source code. The total indexed information size is almost 3 TB of 27 TB source code, which is quite small in size as compare to Benjamin [4] technique (3 times of original system).

C. RQ2: DCCD vs State-of-the-art Methods

In a comparison of old preprocessing, indexing and clone detection techniques, our approach is much faster as shown in Table II. In Benjamin [4] technique, they used 100 Google machines to build 73 million LOC source code in 3 hours on Intel Xeon processor with 3GB RAM, which was 86 times faster than CCFinder [5] technique. In our approach, we just spend about 45 minutes to build the same amount of source code (73 million LOC) using 5 distributed systems (Intel i5, 16GB). CCFinder processed 400 MLOC on 80 machines (Pentium-4:3Ghz, 1GB RAM) in 51 hours, in our case we processed 400 MLOC in almost 5 hours on a single machine (Intel i5, 16GB). CCFinder [5] and Benjamin [4] just detected Type-1 and Type-2 clones. The most important achievement is that while they only detect Type-1, Type-2 clones. We detect all three types of clones in less time with big code, even though it was very challenging.

TABLE II: Build Index Speed Comparison

Technique	Machines	LOC(Million)	Time(Hours)	Clone Types
CCFinder	80	400	51	T-1, T-2
Benjamin	10	73	3	T-1, T-2
DCCD	5	3300	6	T-1, T-2, T-3

D. Batch Clone Detection

This case study shows that our feature extraction approach in the case of batch clone detection produces very good results. We used two open source projects of C and Java language as test projects. In the comparison of our technique (DCCD) with others techniques, i.e. Suffix-tree and Benjamin, the execution time of our approach is quite fast as shown in Table III. For each of the testing systems, DCCD is fast and meanwhile, it detects 3 types of clones, which is not possible in other 2 techniques.

TABLE III: Clone Detection Execution Time Comparison

Techniques	Jabref	Linux-Kernel	Clone Types
Suffix-tree	7.3 sec	166 min 13 sec	T-1, T-2
Benjamin	6.7 sec	47 min 29 sec	T-1, T-2
DCCD	5.2 sec	20 min 40 sec	T-1, T-2, T-3

E. DCCD (Distributed Code Clone Detection)

For the scalability and performance evaluation of our platform & algorithm to a large code base. Clone detection was performed through MapReduce, which retrieves all clones and calculates clone coverage for all project files in the index. In addition, to evaluate the scalability for ultra large code scale systems, we selected some subject systems including, e.g. Linux 2.6.33, Harvey, Cinder, PostgreSQL, OpenCV and Arduino. The detection processed 16.5 MLOC of C /C++ code of 37,398 files. In our experiment, we applied different filters of code clone detection method on every test project, the graphical representation have been shown in Fig 5.

TABLE IV: Detection Results Against their Files, LOC, Time and Clone Types

Test Projects	Total Files	LOC	Detection Time	Type-1 Clones	Type-2 Clones	Type-3 Clones
Linux 2.6.33	25,717	11,267,973	3 hours 20 min	1867	1443	3031
Harvey	3,761	1,307,197	22 min 37 sec	1460	546	917
Cinder	2,955	1,180,935	14 min 25 sec	1127	157	307
PostgreSQL	1,906	1,332,103	17 min 15 sec	889	356	373
OpenCV	2,379	1,141,501	13 min 48 sec	273	63	88
Arduino	680	277,710	9 min 33 sec	62	25	69

TABLE V: Retrieved Results Against Each Filter

C&C++ repository info: (Code: 16 TB, Files: 493 Million, LOC:207 billion)

Test Projects	Filter-1	Filter-2	Filter-3	Filter-4
Linux 2.6.33	99,031	95,471	78,570	66,034
Harvey	82,013	51,492	35,301	31,101
Cinder	57,788	37,031	25,701	13,876
PostgreSQL	40,371	27,096	19,811	14,503
OpenCV	17,359	10,782	8,703	5,609
Arduino	10,093	7,182	3,935	1,808

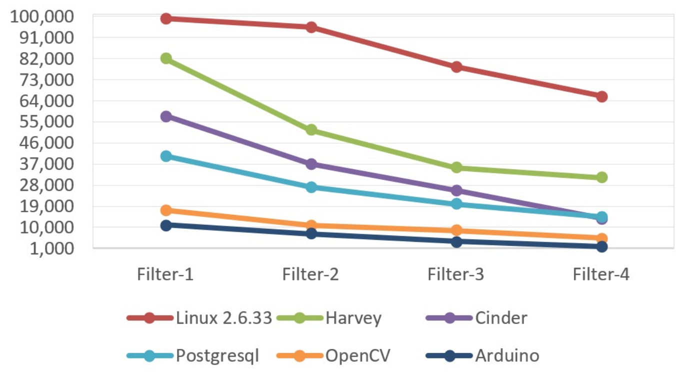


Fig. 5: Graphical representation of Table V

Table IV shows the test projects against their total number of project files, LOC, detection time and a number of clones of Type-1, Type-2, Type-3 in each testing project.

F. RQ3: Detection of Full Level and Partial Level Similarities

During full application similarity detection, we detected almost all cloning files from our big code base repository against our subject system entitled *Linux-Kernel* as the results have been shown Table IV. In the case of partial similarity detection, our approach successfully found the systems from code base, which are sharing some source files or part of their source codes. Both full and partial level of similarity detection basically requires finding the similar code fragments in source code.

G. RQ4: System Scalability and Efficiency

The execution time actually scales with the size of processed code (LOC) by a tool. As we used MapReduce model for parallel and distributing processing in a cluster, which increase the scalability and efficiency of our approach, meanwhile it is very cost effective and affordable solution. The DCCD approach is able to scale 324 billion LOC, as it has been tested. The DCCD execution time of indexing and detection is quite faster than other techniques [6]. It is the only technique which has detected Type-1, Type-2 and Type-3 clones from big code source repositories of 27 TB. As the cluster was also used for other purposes, so we measured the time based on its overall load. The results from these case studies show that our proposed approach is very capable of supporting distributed code clone detection and batch clone detection in real time environment for big code bases.

IV. RELATED WORK

Recently, many clone detection approaches have been developed for small scale and large scale systems [7] [6] [8] [9] [10] [11] [12] [13] [14] [15] [16] [17] [18], which detects clones on different level of granularities. Bundle of code clone detection techniques even for Java Bytecode [19] already have been proposed and implemented [20] [21] [22]. Their results and method have been employed by code clone management tools [23]. Some methods were implemented and embedded in programming platforms, for example, SimEclipse (clone detector plugin), Visual Studio and so on [24]. NICAD [25] by Roy is also considered as hybrid approch. NICAD technique uses *Longest Common Subsequence* algorithm to compare lines of source code [26]. The lexical approaches include CCFinder [5] by Kamiya, CP-Miner by Zhenmin, Boreas by Yong and Yao, FRISC by Murakami, and CDSW by Murakami [20] [27]. The token matching suffix tree based algorithm was used by CCFinder to find out all similar token sequences. Recently there is a tool SourcererCC [24], which performs code clone detection to big code but data set is not big as compared to DCCD, and it is not distributed. Meanwhile we explored different syntactical approaches, i.e CloneDr et al., Wahler et al., Koschke et al. [27], Jiang et al. Hotta et al. Mayrand et al, Kontogiannis, Kodhai, et al. Abdul-El-Hafiz, Kanika et al. [20] [2]. Metric-based [28] need parser to obtain values of metrics, and even it is possible that two code fragments having same metric values maybe not similar code fragments. CONQAT [29] [30] considered as hybrid, clones are detected in main three phases. During our review study in this field, we explored some important semantic techniques, i.e. Komondoor and Horwitz (PDGs using CodeSurfer), Duplix (PDGs) by Krinke et al., GPLAG (PDGs using CodeSurfer) by Liu et al, Higo and Kusumoto (PDGs using CodeSurfer), ConQAT (Suffix-tree-based, Token) [4], Funaro (AST) and Agrawal (Tokens) [20]. PDG based techniques are not scalable for large scale systems [23] [29], because it needs a PDG generator and graph matching, which is little bit expensive.

V. Conclusion and Future Work

In this paper, we proposed a Distributed Code Clone Detection (DCCD) technique for token-based code clone detection, which retrieves clones in an efficient way. It exploits an index of source code to achieve the scalability and maintenance of large scale project repositories for big code. The index of 27 TB source code (C, C++, Java, JavaScript) has been built in less than a month. To the best of our knowledge, this is the first approach which has been implemented for large scale systems, which is have been experimented on 27 TB of source code, meanwhile it detects all 3 types of clones very efficiently, especially Type-3 which was very challenging in large scale system. DCCD has been achieved a high accuracy (87%) rate in clone detection for large scale system. DCCD can be adopt at industry level for the detection of clones in big code, which is easily extendible and cost effective. For future concerns, our next target is to extend the current work and continue building a big index for other programming languages i.e. Python, Php, Xml, C#, Vb, Cobol, Text, Sql, Matlab, Ruby, Ada. Meanwhile we are considering to add additional functionalities related to vulnerabilities detection in systems.

VI. Acknowledgment

This research was supported by Beijing National Research Center for Information Science and Technology (BNRist), and National Natural Science Foundation of China under Grant Nos. 90818021, 9071803.

References

[1] C. K. Roy and J. R. Cordy, "An empirical study of function clones in open source software," in *Reverse Engineering, 2008. WCRE'08. 15th Working Conference on.* IEEE, 2008, pp. 81–90.

[2] E. Choi, N. Yoshida, T. Ishio, K. Inoue, and T. Sano, "Extracting code clones for refactoring using combinations of clone metrics," in *Proceedings of the 5th International Workshop on Software Clones*. ACM, 2011, pp. 7–13.

[3] J. Svajlenko, J. F. Islam, I. Keivanloo, C. K. Roy, and M. M. Mia, "Towards a big data curated benchmark of inter-project code clones," in *Software Maintenance and Evolution (ICSME), 2014 IEEE International Conference on.* IEEE, 2014, pp. 476–480.

[4] B. Hummel, E. Juergens, L. Heinemann, and M. Conradt, "Index-based code clone detection: incremental, distributed, scalable," in *Software Maintenance (ICSM), 2010 IEEE International Conference on.* IEEE, 2010, pp. 1–9.

[5] 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*, vol. 28, no. 7, pp. 654–670, 2002.

[6] H. Sajnani, V. Saini, J. Svajlenko, C. K. Roy, and C. V. Lopes, "Sourcerercc: Scaling code clone detection to big-code," in *Software Engineering (ICSE), 2016 IEEE/ACM 38th International Conference on.* IEEE, 2016, pp. 1157–1168.

[7] J. Svajlenko and C. K. Roy, "A machine learning based approach for evaluating clone detection tools for a generalized and accurate precision," *International Journal of Software Engineering and Knowledge Engineering*, vol. 26, no. 09n10, pp. 1399–1429, 2016.

[8] S. Charalampidou, A. Ampatzoglou, A. Chatzigeorgiou, and P. Avgeriou, "Assessing code smell interest probability: A case study," 2017.

[9] J. Svajlenko and C. K. Roy, "Fast and flexible large-scale clone detection with cloneworks," in *Proceedings of the 39th International Conference on Software Engineering Companion.* IEEE Press, 2017, pp. 27–30.

[10] T. Hatano and A. Matsuo, "Removing code clones from industrial systems using compiler directives," in *2017 IEEE/ACM 25th International Conference on Program Comprehension (ICPC)*, May 2017, pp. 336–345.

[11] S. Alam, Z. Qu, R. Riley, Y. Chen, and V. Rastogi, "Droidnative: Automating and optimizing detection of android native code malware variants," *computers & security*, vol. 65, pp. 230–246, 2017.

[12] Y. Yuki, Y. Higo, and S. Kusumoto, "A technique to detect multi-grained code clones," in *Software Clones (IWSC), 2017 IEEE 11th International Workshop on.* IEEE, 2017, pp. 1–7.

[13] S. Kim, S. Woo, H. Lee, and H. Oh, "Vuddy: A scalable approach for vulnerable code clone discovery," in *Security and Privacy (SP), 2017 IEEE Symposium on. IEEE*, 2017.

[14] F. Lyu, Y. Lin, J. Yang, and J. Zhou, "Suidroid: An efficient hardening-resilient approach to android app clone detection," in *Trustcom/BigDataSE/I SPA, 2016 IEEE.* IEEE, 2016, pp. 511–518.

[15] J. Svajlenko and C. K. Roy, "Evaluating clone detection tools with bigclonebench," in *Software Maintenance and Evolution (ICSME), 2015 IEEE International Conference on.* IEEE, 2015, pp. 131–140.

[16] Y. Dang, D. Zhang, S. Ge, R. Huang, C. Chu, and T. Xie, "Transferring code-clone detection and analysis to practice," in *Proceedings of the 39th International Conference on Software Engineering: Software Engineering in Practice Track.* IEEE Press, 2017, pp. 53–62.

[17] Y. Hu, Y. Zhang, J. Li, and D. Gu, "Binary code clone detection across architectures and compiling configurations," in *Proceedings of the 25th International Conference on Program Comprehension.* IEEE Press, 2017, pp. 88–98.

[18] C. Ragkhitwetsagul and J. Krinke, "Using compilation/decompilation to enhance clone detection," in *Software Clones (IWSC), 2017 IEEE 11th International Workshop on.* IEEE, 2017, pp. 1–7.

[19] D. Yu, J. Wang, Q. Wu, J. Yang, J. Wang, W. Yang, and W. Yan, "Detecting java code clones with multi-granularities based on bytecode," in *2017 IEEE 41st Annual Computer Software and Applications Conference (COMPSAC)*, vol. 01, July 2017, pp. 317–326.

[20] A. Sheneamer and J. Kalita, "A survey of software clone detection techniques," *International Journal of Computer Applications*, pp. 0975–8887, 2016.

[21] T. Zhang and M. Kim, "Automated transplantation and differential testing for clones," in *Proceedings of the 39th International Conference on Software Engineering.* IEEE Press, 2017, pp. 665–676.

[22] H. Sajnani, V. Saini, and C. Lopes, "A parallel and efficient approach to large scale clone detection," *Journal of Software: Evolution and Process*, vol. 27, no. 6, pp. 402–429, 2015.

[23] X. Cheng, H. Zhong, Y. Chen, Z. Hu, and J. Zhao, "Rule-directed code clone synchronization," in *Program Comprehension (ICPC), 2016 IEEE 24th International Conference on.* IEEE, 2016, pp. 1–10.

[24] V. Saini, H. Sajnani, J. Kim, and C. Lopes, "Sourcerercc and sourcerercc-i: tools to detect clones in batch mode and during software development," in *Proceedings of the 38th International Conference on Software Engineering Companion.* ACM, 2016, pp. 597–600.

[25] C. K. Roy and J. R. Cordy, "Nicad: Accurate detection of near-miss intentional clones using flexible pretty-printing and code normalization," in *Program Comprehension, 2008. ICPC 2008. The 16th IEEE International Conference on.* IEEE, 2008, pp. 172–181.

[26] J. R. Cordy and C. K. Roy, "The nicad clone detector," in *Program Comprehension (ICPC), 2011 IEEE 19th International Conference on.* IEEE, 2011, pp. 219–220.

[27] A. Walenstein, R. Koschke, and E. Merlo, "Duplication, redundancy, and similarity in software: Summary of dagstuhl seminar 06301," *Dagstuhl, Germany, Dagstuhl*, 2006.

[28] M. S. Aktas and M. Kapdan, "Structural code clone detection methodology using software metrics," *International Journal of Software Engineering and Knowledge Engineering*, vol. 26, no. 02, pp. 307–332, 2016.

[29] Y. Ueda, T. Kamiya, S. Kusumoto, and K. Inoue, "Gemini: Maintenance support environment based on code clone analysis," in *Software Metrics, 2002. Proceedings. Eighth IEEE Symposium on.* IEEE, 2002, pp. 67–76.

[30] M. S. Uddin, V. Gaur, C. Gutwin, and C. K. Roy, "On the comprehension of code clone visualizations: A controlled study using eye tracking," in *Source Code Analysis and Manipulation (SCAM), 2015 IEEE 15th International Working Conference on.* IEEE, 2015, pp. 161–170.