SourcererCC: Scalable and Accurate Clone Detection

Hitesh Sajnani, Vaibhav Saini, Chanchal K. Roy, and Cristina Lopes

Abstract Clone detection is an active area of research. However, there is a marked lack in clone detectors that scale to very large repositories of source code, in particular for detecting near-miss clones where significant editing activities may take place in the cloned code. SourcererCC was developed as an attempt to fill this gap. It is a widely used token-based clone detector that targets three clone types, and exploits an index to achieve scalability to large inter-project repositories using a standard workstation. SourcererCC uses an optimized inverted-index to quickly query the potential clones of a given code block. Filtering heuristics based on token ordering are used to significantly reduce the size of the index, the number of code-block comparisons needed to detect the clones, as well as the number of required token-comparisons needed to judge a potential clone. In the evaluation experiments, SourcererCC demonstrated both high recall and precision, and the ability to scale to a large inter-project repository (250MLOC) even using a standard workstation. This chapter reflects on some of the principle design decisions behind the success of SourcererCC and also presents an architecture to scale it horizontally.

1 Introduction

With the amount of source code increasing steadily, large-scale clone detection has become a necessity. Large code bases and repositories of projects have led to several new use cases of clone detection including mining library candidates [11], detecting similar mobile applications [3], license violation detection [14], reverse engineering product lines [8], finding the provenance of a component [5], and code search [12].

H. Sajnani (✉) · V. Saini
Microsoft, Redmond, USA
e-mail: hsajnani@uci.edu

C. K. Roy
University of Saskatchewan, Saskatoon, Canada

C. Lopes
University of California, Irvine, USA

K. Inoue and C. K. Roy (eds.), *Code Clone Analysis*,
https://doi.org/10.1007/978-981-16-1927-4_4

While presenting new opportunities for the application of clone detection, these modern use cases also pose scalability challenges.

1.1 Motivation

To further illustrate the problem and its scale in practice, consider a real-life scenario where a retail banking software system is maintained by Tata Consultancy Services (TCS). A team at TCS deployed the banking system for many different banks (clients) and maintained a separate code base for each of these banks. After following this practice for a while, they decided to form a common code base for all these banks to minimize expenses occurring due to (i) duplicated efforts to deliver the common features and (ii) separately maintaining existing common parts of different code bases.

As part of this bigger goal, the team decided to first identify common code blocks across all the code bases. In order to assess the feasibility of using clone detection tools for this task, the team[1] ran CloneDR, an AST-based commercial clone detection tool on ML0000, a single COBOL program consisting of 88K LOC. The clone detection process took around 8 h on an IBM T43 Thinkpad default specification[2] machine. Each bank's code base (8 of them) ran into multi-million lines of code spanning across thousands of such COBOL programs in different dialects, posing a major scalability challenge.

This situation at TCS is not unique and in fact represents the state of many companies in the service industry that are now moving away from the greenfield development model and adopting the packaging model to build and deliver software. In fact, as Cordy points out, it is a common practice in industry to clone a module and maintain it in parallel [4]. Similarly in open source development, developers often clone modules or fork projects to meet the needs of different clients, and may need large-scale clone detectors to merge these cloned systems toward a product-line style of development.

While the above use cases are more pertinent to industry, researchers are also interested in studying cloning in large software ecosystems (e.g., Debian), or in open-source development communities (e.g., GitHub) to assess its impact on software development and its properties. However, very few tools can scale to the demands of clone detection in very large code bases [20]. For example, Kim and Notkin [13] reflected how they wanted to use clone detection tools for doing origin analysis of software files but were constrained by its speed due to n-to-n file comparison. In his work on using clone detection to identify license violations, Koschke [14] reflects the following: "Detecting license violations of source code requires to compare a suspected system against a very large corpus of source code, for instance, the Debian source distribution. Thus, code clone detection techniques must scale in

[1] The author was part of the team that carried out the analysis.

[2] i5 processor, 8 GB RAM, and 500 GB disk storage.

terms of resources needed". In 2014, Debian had 43,000[3] software packages and approximately 323 million lines of code. In one of their studies to investigate cloning in FreeBSD, Livieri et al. [16] motivate the need for scalable code clone detection tools as follows: "Current clone detection tools are incapable of dealing with a corpus of this size, and might either take literally months to complete a detection run, or might simply crash due to lack of resources".

1.2 Challenges

While a few novel algorithms [10, 16] in the last decade demonstrated scalability, they do not support Type-3 near-miss clones, where minor to significant editing activities might have taken place in the copy/pasted fragments. These tools therefore miss a large portion of the clones, since there are more number of Type-3 clones in the repositories than the other types. Furthermore, the ability to detect Type-3 clones is most needed in large-scale clone detection applications [18].

Many techniques have also been proposed to achieve a few specific applications of large-scale clone detection [3, 14], however, they make assumptions regarding the requirements of their target domain to achieve scalability, for example, detecting only file-level clones to identify copyright infringement, or detecting clones only for a given block (clone search) in a large corpus. These domain-specific techniques are not described as general large-scale clone detectors, and face significant scalability challenges for general clone detection.

The scalability of clone detection tools is also constrained by the computational nature of the problem itself. A fundamental way of identifying if two code blocks are clones is to measure the degree of similarity between them, where similarity is measured using a similarity function. A higher similarity value indicates that code blocks are more similar. Thus we can consider pairs of code blocks with high similarity value as clones. In other words, to detect all the clones in a system, each code block has to be compared against every other code block (also known as candidate code blocks), bearing a prohibitively $O(n^2)$ time complexity. Hence, it is an algorithmic challenge to perform this comparison in an efficient and scalable way. This challenge, along with modern use cases and today's large systems, makes large-scale code clone detection a difficult problem.

The above challenges can be characterized in the form of the following research questions.

Research Question 1. [Design]—How can we be more robust to modifications in cloned code to detect Type-3 clones?

Research Question 2. [Computational Complexity]—How can we reduce the $O(n^2)$ candidate comparisons to $O(c.n)$, where $c << n$?

Research Question 3. [Engineering]—How can we make faster candidate comparisons without requiring much memory?

[3] https://en.wikipedia.org/wiki/Debian.

Research Question 1 has a direct implication on the accuracy of the clone detection technique, and Research Questions 2 and 3 focus on improving the scalability and efficiency of the clone detection technique.

SourcererCC [19] addresses the above challenges leading to an accurate and fast approach to clone detection that is both scalable to very large software repositories and robust against code modifications.

2 SourcererCC

The core idea of SourcererCC is to build an optimized index of code blocks and compare them using a simple and fast bag-of-tokens[4] strategy which is resilient to Type-3 changes (Research Question 1). Several filtering heuristics are used to reduce the size of the index, which significantly reduces the number of code block comparisons to detect the clones. SourcererCC also exploits the ordering of tokens in a code block to measure a live upper-bound on the similarity of code blocks in order to reject or accept a clone candidate with minimal token comparisons (Research Question 2 and 3).

2.1 Bag-of-Tokens Model

SourcererCC represents a code block using a bag-of-tokens model where tokens are assumed to appear independently of one another and their order is irrelevant. The idea is to transform code blocks in a form that enables SourcererCC to detect clones that have different syntax but similar meaning. Moreover, this representation also filters out code blocks with specified structure patterns. Since SourcererCC matches tokens and not sequences or structures, it has a high tolerance to minor modifications, making it effective in detecting Type-3 clones, including clones where statements are swapped, added, and/or deleted.

The overlap similarity measure simply computes the intersection between the code fragments by counting the number of tokens shared between them. The intuition here is simple. *If two code fragments have many tokens in common, then they are likely to be similar to some degree.*

It is interesting to note that such a simple strategy could prove to be so effective in a complex software engineering task of identifying code clones. The primary reason for the effectiveness of bag-of-tokens and overlap similarity measure is rooted in the program vocabulary used by the developers while writing code. While programming languages in theory are complex and powerful, the programs that real people write are mostly simple and rather repetitive and similar [9]. This similarity is manifested in the source code in the form of tokens, and particularly in identifiers. In source code,

[4] Similar to the popular bag-of-words model [22] in Information Retrieval.

identifiers (e.g., names of variables, methods, classes, parameters, and attributes) account for approximately more than 70% of the linguistic information [6]. Many researchers have concluded that identifiers reflect the semantics and the role of the named entities they are intended to label [1, 7, 15]. Therefore, code fragments having similar semantics are likely to have similarities in their identifiers. Furthermore, oftentimes, during copy-paste-modify practice, developers preserve identifier names as they reflect the underlying functionality of the code that is copied. They seem to be aware of the fact that different names used for the same concept or even identical names used for different concepts reflect misunderstandings and foster further misconceptions [6]. As a result, while copied fragments are edited to adapt to the context in which they are copied, they often have enough syntactical similarity associated with the original fragment. This similarity is effectively captured by the bag-of-tokens model in conjunction with the overlap similarity measure.

Of course, there are scenarios when programmers may deliberately obfuscate code to conceal its purpose (security through obscurity) or its logic, in order to prevent tampering, deter reverse engineering, hide plagiarism, or as a puzzle or recreational challenge for someone reading the source code. The simple bag-of-tokens model of SourcererCC may not be effective in detecting clones in such cases. Other tools like Deckard that rely on AST, or NiCad that uses heavy normalization, may be effective under such scenarios.

2.2 Filtering Heuristics to Reduce Candidate Comparisons

In order to detect all clone pairs in a project or a repository, the above approach of computing the similarity between two code blocks can simply be extended to iterate over all the code blocks and compute pairwise similarity for each code block pair. For a given code block, all the other code blocks compared are called candidate code blocks or candidates in short. While the approach is very simple and intuitive, it is also subject to a fundamental problem that prohibits scalability—$O(n^2)$ time complexity. Figure 1 describes this by plotting the number of total code blocks (X-axis) versus the number of candidate comparisons (Y-axis) in 35 Apache Java projects. Note that the granularity of a code block is taken as a method. Points denoted by ∘ show how the number of candidates compared increases quadratically[5] with the increase in the number of methods. SourcererCC uses advanced index structures and filtering heuristics—*sub-block overlap filtering and token position filtering*—to significantly reduce the number of candidate comparisons during clone detection. These heuristics are inspired by the work of Chaudhuri et al. [2] and Xiao et al. [21] on efficient set similarity joins in databases. Sub-block overlap filtering follows an intuition that when two sets have a large overlap, even their smaller subsets should overlap. Since

[5] The curve can also be represented using $y = x(x-1)/2$ quadratic function where x is the number of methods in a project and y is the number of candidate comparisons carried out to detect all clone pairs.

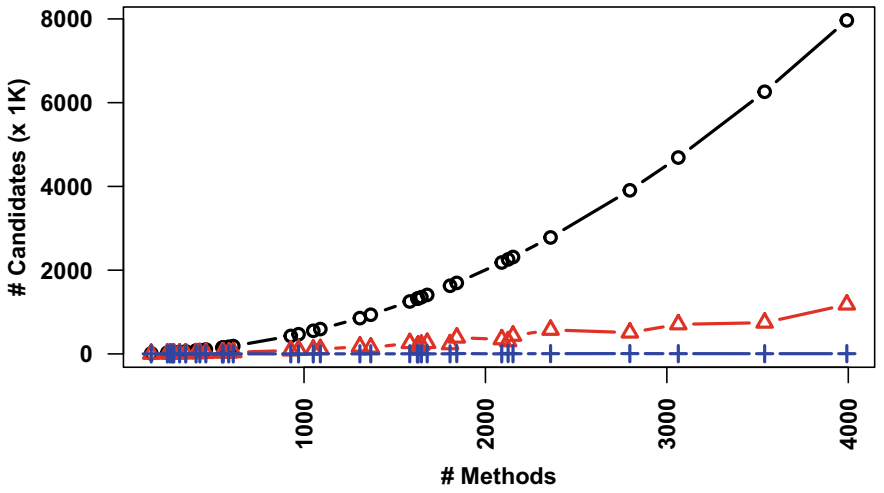

Fig. 1 Growth in the number of candidate comparisons with the increase in the number of code blocks. ∘ show quadratic increase in candidate comparisons; △ denote the number of candidate comparisons after applying the sub-block overlap filtering; + denote the number of candidate comparisons after applying the token position filtering

we represent code blocks as bag-of-tokens (i.e., a multiset), we can extend this idea to code blocks, i.e., when two code blocks have a large overlap, even their smaller sub-blocks should overlap. This constraint allows one to reduce the number of candidate comparisons by eliminating candidates that do not share a similarity in their sub-blocks. Revisiting Fig. 1, the points denoted by △ show the number of candidate comparisons after applying the sub-block overlap filtering. The large difference from the earlier curve (∘) shows the impact of filtering in eliminating candidate comparisons. It turns out that if the tokens in the code block are further arranged to follow a pre-defined order (e.g., order of popularity of tokens in the corpus), we can further reduce the number of token and candidate comparisons by computing a safe upper-bound (without violating the correctness). This filtering is termed token position filtering. The points denoted by + in Fig. 1 show the number of candidate comparisons after applying the token position filtering. The reduction is so significant that empirically on this dataset, the function seems to be *near-linear*. This is a massive reduction in the number of comparisons when compared to the quadratic number of comparisons shown earlier without any filtering.

3 Distributed SourcererCC: Scaling SourcererCC Horizontally

SourcererCC advances the state of the art in code clone detection tools that can scale vertically using high-power CPUs and memory added to a single machine. While this approach works well in most of the cases, in certain scenarios using vertical scalable approaches may not be feasible as they are bounded by the amount of data that can fit into the memory of a single machine. In such scenarios, timely computing clones in ultra-large datasets are beyond the capacity of a single machine.

Under such scenarios, efficient parallelization of the computation process is the only feasible option. Previously, research in this direction was limited due to the lack of the availability of resources and the cost of setting up the infrastructure. But the recent developments in the field of cloud computing and the availability of low-cost infrastructure services like Amazon Web Services (AWS), Azure, and Google Cloud have enabled the research in this area.

However, it is important to note that simply dividing the input space and parallelizing the clone detection operation do not solve the problem, because running tools on projects individually, and then combining the results in the later step, would lead to a collection of common clones, but would not identify clones across division boundaries [16]. Thus, efficient parallelization of the computation process is necessary.

SourcererCC's extensible architecture can be easily adapted to horizontally scale to multiple processors and efficiently detect the first three types of clones on large datasets preserving the same detection quality (recall and precision). We call this extension of SourcererCC Distributed SoucererCC or SoucererCC-D.

SourcererCC-D operates on a cluster of nodes by constructing the index of the entire corpus that is shared across all the nodes, and then parallelizes the clone searching process by distributing the tasks across all the nodes in the cluster. In order to achieve this, SourcererCC-D follows a standard *Shared disk* (see Fig. 2) or a *Shared memory* (see Fig. 3) architecture style.

A shared disk architecture (SD) is a distributed computing paradigm in which all disks are accessible from all the cluster nodes. While the nodes may or may not have their own private memory, it is imperative that they at least share the disk space. A

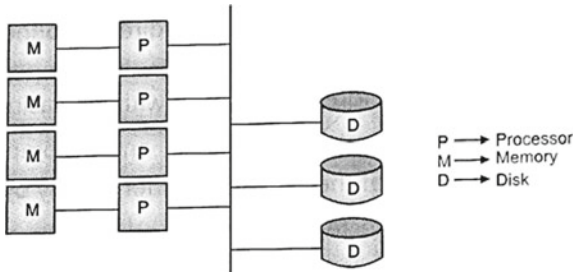

Fig. 2 Shared disk architecture style

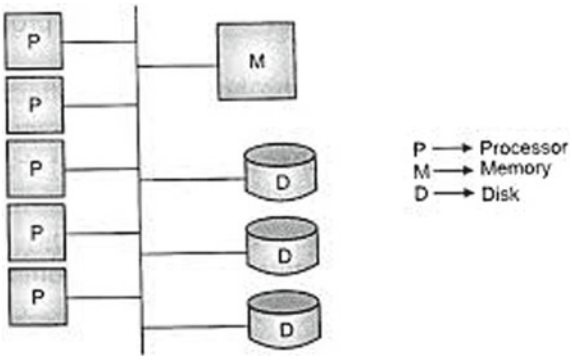

Fig. 3 Shared memory architecture style

shared memory architecture (SM) is a distributed computing paradigm in which the cluster nodes not only share the disks but also have access to global shared memory.

Figure 4 describes SourcererCC-D's clone detection process. Let us assume a cluster of $N + 1$ nodes.[6] Initially, a master node (any one of the cluster nodes) runs the parser on the entire corpus and produces a parsed input file containing code blocks according to the granularity set by the user. Next, the master node runs SourcererCC's Indexer on the parsed input file to build an index. The constructed index, also known as a global index, resides on the shared disk and hence is accessible by all the nodes in the cluster. After the global index is constructed, the master node splits the parsed input file into N different files namely Q_1, Q_2, Q_3,$\ldots Q_N$ and distributes them to each node in the cluster. Each node is now responsible to locally compute clones of code blocks in its respective query file using SourcererCC's Searcher. Note that since each node has access to the global index, it can find all the clones in the entire corpus for its given input, i.e., clones present across other nodes. It is for this reason, nodes must have a shared disk space to store the global index. When all the nodes finish executing the Searcher, all the clones in the corpus are found.

Note that in the above design, while the search phase is distributed and happens in parallel, the index construction phase is not parallelized (only the master node constructs the index). However, this hardly impacts the overall clone detection performance because we found that index construction takes less than 4% of the total time to detect clones.

SourcererCC-D can be deployed on in-house clusters, cloud services like AWS, Azure, or even in-house multi-processor machines. The ability to scale to multiple machines enables SourererCC to be effectively used for ultra-large datasets (e.g., entire corpus of GitHub) as demonstrated in [17].

[6] In case of a single high-performance multi-processor machine, $N + 1$ is the number of processors available on that machine.

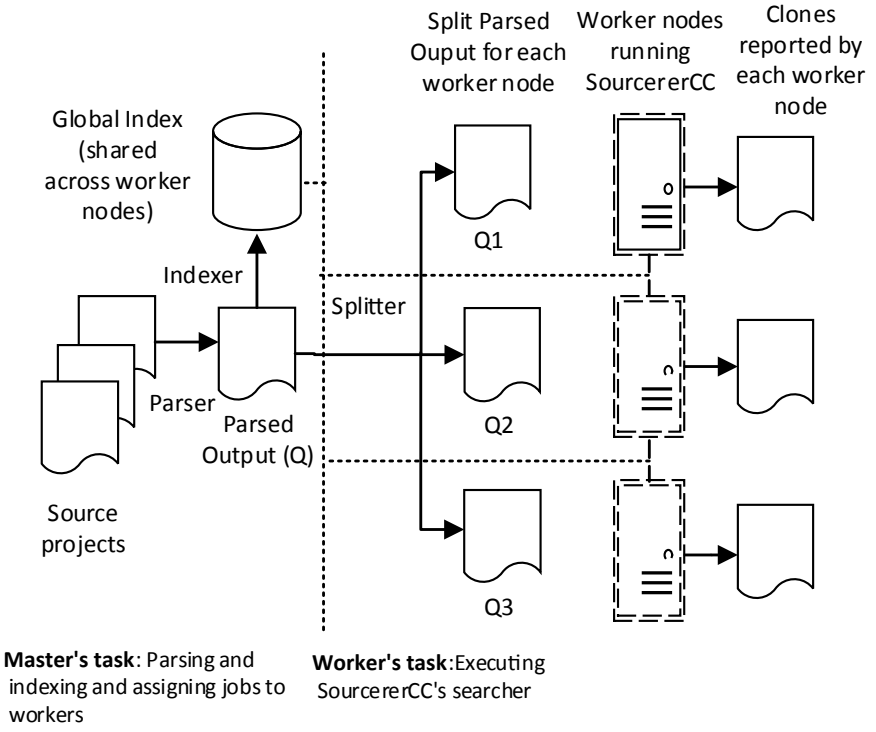

Fig. 4 SourcererCC-D's clone detection process

4 Lessons Learned During Implementation

There were many lessons learned during the design, development, and testing of SourcererCC. While these lessons are not new, this section reflects upon them, as they played an important role in the successful development and adoption of the SourcererCC tool in the academic community. We believe tool designers could benefit from our experience and reflection.

Everything breaks at scale. One of the key lessons during SourcererCC's development can be aptly described in a phrase—"Everything breaks at scale, so expect the unexpected". We realized that at scale, we cannot test for every error. As a result, we used assertions and exception handlers for things that can't happen. We added diagnostic code, logging, and tracing to help explain what is going on at run-time, especially when we ran into problems during development. The philosophy—if this failed, look for what else can fail—played a very important role during SourcererCC's development.

Fault Tolerance. During the initial stages of the development, SourcererCC crashed at times while running on large datasets after several hours of execution due to unexpected reasons. Since SourcererCC did not have the mechanism to pre-

serve its execution state during that time, such failures resulted in a loss of several hours of computation time and effort, not to mention the frustration that comes along. Based on these experiences, we realized that SourcererCC's exception handler must preserve its state of the execution (i.e., keeping track of how much data is already processed), so when interrupted, SourcererCC's execution can resume correctly from the point of failure at a later time. The necessity of logging how much data is processed by the tool is an important lesson that we learned the hard way.

Memory Leaks. SourcererCC is written in Java programming language which has its own garbage collection mechanism. However, we encountered bugs related to memory leaks while testing SourcererCC on large datasets. What I learned from debugging memory leak issues is that while there are simple solutions to detect and deal with memory leaks (e.g., logging the size of your data structures when you modify them, and then searching the logs for data structures that grow beyond a reasonable size), a tool is undoubtedly a big help. In the absence of the right tools, debugging such issues could take unreasonable time and effort. We were able to resolve these issues much faster using open-source tools like VisualVM[7] and Profiler4J.[8]

The problem could be in the data too. Oftentimes when we noticed anomalies in the execution of SourcererCC, we thought that the issue would be in the code. However, it was not unusual to find issues with either the input data or our assumptions about the input data. As a result, we realized that it is always useful to check for data consistency and integrity even before any experimentation.

Tuning parameters to optimize SourcererCC's performance. SourcererCC has a few parameters (e.g., similarity threshold, tokenization strategies, and minimum size threshold of a code block) that had to be tuned to optimize for accuracy, scalability, and efficiency. This resulted in countless experiments, and keeping track of these experiments and their settings posed a severe challenge.

To do this exercise systematically, we adopted the following process that indeed turned out to be very effective.

We created a smaller dataset for parameter tuning experiments. Apart from the smaller size, this dataset had characteristics very similar to the large datasets on which SourcererCC is intended to be used. Executing SourcererCC on a smaller dataset took less time, thus giving us more freedom to experiment.

In order to better keep track of SourcererCC's performance on different parameter configurations, we created a SourcererCC revision (using Git) for each configuration of parameters. This not only enabled us to run several experiments in parallel but also helped to easily switch back-and-forth across various parameter configurations.

To summarize, creating SourcererCC's revisions for various parameter configurations and running them in parallel on a smaller dataset greatly reduced the turnaround time for performing experiments to tune SourcererCC.

SourcererCC is publicly available and actively maintained. It can be downloaded from https://github.com/Mondego/SourcererCC.

[7] https://visualvm.java.net/.

[8] http://profiler4j.sourceforge.net/.

5 Going Forward

Code Clone detection research has come a long way in the last couple of decades. We conclude by identifying some of the relevant areas that might shape the future research in this field. There are many tools available for clone detection. In contrast, there are relatively few tools that help in removing or effectively managing clones. Identifying various means of eliminating harmful clones through automated tool support is an interesting venue to explore in the future. Large-scale clone detection is often faced with the challenge of how to make sense of the large data produced by the clone detection tools. Visual and interactive representations of the output to reinforce human cognition and produce actionable insight is another useful direction for the future. The utility of clone detection is not just limited to source code. Clone detection in other software artifacts, including models, bug-reports, requirement documents, and binaries, is turning out to be a necessity for several use cases. For example, the ability to detect clones in software binaries is necessary for effectively detecting Malwares and License Infringement. Therefore, extending code clone detection research to other software artifacts is a promising area for the future. Clone research should also focus on clone management by (i) identifying and prioritizing the clones that are of interest to the developers for a given task, (ii) helping developers pro actively assess the negative consequence of cloning, and (iii) categorizing clones as harmful and harmless after detection. With the several new use cases of clone detection emerging, a reorientation of research focus toward application-oriented clone detection might be useful. In many cases, state-of-the-art clone detection tools do not behave well for these specific use cases. These observations point to the new research opportunities to enhance clone detection technologies. Moreover, use case-specific benchmarking to evaluate various tools and techniques might be another area to focus on in the future.

References

1. C. Caprile, P. Tonella, Nomen est omen: analyzing the language of function identifiers, in *Reverse Engineering. Proceedings. Sixth Working Conference on* (1999), pp 112–122. https://doi.org/10.1109/WCRE.1999.806952
2. S. Chaudhuri, V. Ganti, R. Kaushik, A primitive operator for similarity joins in data cleaning, in *Proceedings of the 22nd International Conference on Data Engineering, IEEE Computer Society, Washington, DC, USA, ICDE '06* (2006), pp 5. https://doi.org/10.1109/ICDE.2006.9
3. K. Chen, P. Liu, Y. Zhang, Achieving accuracy and scalability simultaneously in detecting application clones on android markets, in *Proceedings of the 36th International Conference on Software Engineering, ACM, New York, NY, USA*, (ICSE 2014), pp 175–186
4. J.R. Cordy, Comprehending reality-practical barriers to industrial adoption of software maintenance automation, in *Program Comprehension, 2003. 11th IEEE International Workshop on* (IEEE, 2003), pp 196–205
5. J. Davies, D. German, M. Godfrey, A. Hindle, Software Bertillonage: finding the provenance of an entity, in *Proceedings of MSR* (2011)

6. F. Deissenboeck, M. Pizka, Concise and consistent naming. Softw. Qual. J. **14**(3), 261–282. https://doi.org/10.1007/s11219-006-9219-1
7. L. Guerrouj, Normalizing source code vocabulary to support program comprehension and software quality, in *Software Engineering (ICSE), 2013 35th International Conference on*(2013), pp 1385–1388. https://doi.org/10.1109/ICSE.2013.6606723
8. A. Hemel, R. Koschke, Reverse engineering variability in source code using clone detection: a case study for linux variants of consumer electronic devices, in *Proceedings of Working Conference on Reverse Engineering* (2012), pp 357–366
9. A. Hindle, E. T. Barr, Z. Su, M. Gabel, P. Devanbu, On the naturalness of software, in *Proceedings of the 34th International Conference on Software Engineering* (IEEE Press, Piscataway, NJ, USA, ICSE '12, 2012), pp 837–847. http://dl.acm.org/citation.cfm?id=2337223.2337322
10. B. Hummel, E. Juergens, L. Heinemann, M. Conradt, Index-based code clone detection:incremental, distributed, scalable, in *Proceedings of ICSM* (2010)
11. T. Ishihara, K. Hotta, Y. Higo, H. Igaki, S. Kusumoto, Inter-project functional clone detection toward building libraries: an empirical study on 13,000 projects, in *Reverse Engineering (WCRE), 2012 19th Working Conference on* (2012), pp 387–391. https://doi.org/10.1109/WCRE.2012.48
12. I. Keivanloo, J. Rilling, P. Charland, Internet-scale real-time code clone search via multi-level indexing, in *Proceedings of WCRE* (2011)
13. M. Kim, D. Notkin, Program element matching for multi-version program analyses, in *Proceedings of the 2006 International Workshop on Mining Software Repositories* (ACM, New York, NY, USA, MSR '06, 2006), pp 58–64. https://doi.org/10.1145/1137983.1137999
14. R. Koschke, Large-scale inter-system clone detection using suffix trees, in *Proceedings of CSMR* (2012), pp. 309–318
15. D. Lawrie, C. Morrell, H. Feild, D. Binkley, What's in a name? a study of identifiers, in *14th IEEE International Conference on Program Comprehension (ICPC'06)* (2006), pp. 3–12. https://doi.org/10.1109/ICPC.2006.51
16. S. Livieri, Y. Higo, M. Matsushita, K. Inoue, Very-large scale code clone analysis and visualization of open source programs using distributed ccfinder: D-ccfinder, in *Proceedings of ICSE* (2007)
17. C. V. Lopes, P. Maj, P. Martins, V. Saini, D. Yang, J. Zitny, H. Sajnani, J. Vitek, Déjàvu: a map of code duplicates on github, in *Proceedings of the ACM Program Lang 1(OOPSLA)* (2017). https://doi.org/10.1145/3133908
18. C. Roy, M. Zibran, R. Koschke, The vision of software clone management: past, present, and future (keynote paper), in *Software Maintenance, Reengineering and Reverse Engineering (CSMR-WCRE), 2014 Software Evolution Week–IEEE Conference on* (2014), pp. 18–33
19. H. Sajnani, V. Saini, J. Svajlenko, C.K. Roy, C.V. Lopes, Sourcerercc: scaling code clone detection to big-code, in *Proceedings of the 38th International Conference on Software Engineering, Association for Computing Machinery* (New York, NY, USA, ICSE '16, 2016), pp. 1157–1168. https://doi.org/10.1145/2884781.2884877
20. J. Svajlenko, I. Keivanloo, C. Roy, Scaling classical clone detection tools for ultra-large datasets: an exploratory study, in *Software Clones (IWSC), 2013 7th International Workshop on* (2013), pp. 16–22
21. C. Xiao, W. Wang, X. Lin, J. X. Yu, Efficient similarity joins for near duplicate detection, in *Proceedings of the 17th International Conference on World Wide Web* (ACM, New York, NY, USA, WWW '08, 2008), pp. 131–140. https://doi.org/10.1145/1367497.1367516
22. Y. Zhang, R. Jin, Z.H. Zhou, Understanding bag-of-words model: a statistical framework. Int. J. Mach. Learn. Cybern. **1**(1-4), 43–52 (2010). https://doi.org/10.1007/s13042-010-0001-0