On how often code is cloned across repositories

Niko Schwarz
University of Bern

Mircea Lungu
University of Bern

Romain Robbes
University of Chile

***Abstract*—Detecting code duplication in large code bases, or even across project boundaries, is problematic due to the massive amount of data involved. Large-scale clone detection also opens new challenges beyond asking for the provenance of a single clone fragment, such as assessing the prevalence of code clones on the entire code base, and their evolution.**

We propose a set of lightweight techniques that may scale up to very large amounts of source code in the presence of multiple versions. The common idea behind these techniques is to use bad hashing to get a quick answer. We report on a case study, the Squeaksource ecosystem, which features thousands of software projects, with more than 40 million versions of methods, across more than seven years of evolution. We provide estimates for the prevalence of type-1, type-2, and type-3 clones in Squeaksource.

***Index Terms*—Clone detection; Software ecosystems**

I. Introduction

Detecting clones in source code is computationally expensive and does not easily scale up to massive amounts of data such as when analyzing entire software ecosystems [**?**]. On the other hand, counting identical duplicates, even in large amounts of data, is computationally less expensive. It has been shown that indexing source code fragments based on the result of a hashing function, is a promising approach to achieve good performance when large amounts of source code must be handled [1]: The problem of finding snippets of similar source code can be reduced to finding identical hashes, if the hash function is "bad"—generates collisions on similar documents.

The literature defines three types of clones: type-1—identical source code duplication; type-2 clones may feature renames of identifiers; type-3 clones may feature more extensive changes [2]. Current hash-based approaches to clone detection handle only type-1 and type-2. In this paper, we provide hash functions for type-1, type-2, and type-3 clones which exhibit reasonable detection accuracy.

Beyond mere clone detection, exploiting the results is a challenge. Most approaches focus on finding the clones of a given code fragment efficiently. In contrast, we store all hashes of the analyzed corpus in one database. This dedicated infrastructure handles large quantities of clone groups, and allows us to answer cloning-related questions at the level of ecosystems, such as "how much cloning exists between *different* projects?", in contrast to simply searching for the clones related to one fragment. Similar holistic queries include analyzing the successive versions of a given piece of code to detect the origin of a clone among several copies: the version that appeared the first in a software repository is likely the original clone [3].

In this paper, we show how bad hashes, *i.e.,* hashes where similar items collide on the same hash, can identify clones corresponding to each criterion (type-1, type-2, and type-3 clones), and how the analysis must be tailored to the versioning system in use. We use it on the entire history of an open source software ecosystem, Squeaksource which features thousands of projects and tens of thousands of versions in a total of 47 GB of uncompressed source code, or 579 MLOC, to answer holistic queries about clones.

Contributions. The contributions of this paper are threefold:

1) Three lightweight, language-independent, clone detection techniques. Each defines a clone type (1, 2, and 3) which can be detected by bad hashes on source code. The presented techniques scale to entire ecosystems.
2) An evaluation of the three detection techniques in terms of performance on a real-world software ecosystem which demonstrates their scalability
3) Preliminary ecosystem-level results, showing that a large amount of code is duplicated, and that clone groups can feature hundreds of members accross many projects.

II. Related Work

There are two fundamentally different approaches to clone-detection: clustering approaches, and index-based approaches. Traditional clone detection tools compute all pairwise distances of code fragments and then cluster all code fragments based on these distances. A popular example is CCFinder [4]. Livieri *et al.* [5] present an extension of the popular clone detector that is distributed over several machines to improve its scaling, named D-CCFinder, which they used to have 80 machines find all clones in 10.8 GB of source code in 51 hours.

Uddin *et al.* [6] show how hashes can speed the computation of all pairwise distances. In their approach, in a first step, all source code is first hashed, and then in a second step, all pairwise distances are computed from the hashes only. Their approach still requires a third clustering step.

Krinke *et al.* [7] investigated cloned code in 30 projects of the Gnome suite of programs. They found 3096 clone groups (8003 clones in total), and that the probability of clones being copied between systems increased with the size of the clones.

On the other hand, index-based approaches, first suggested by Hummel *et al.* [8], save computation by not having a clustering phase. In their paper, Hummel *et al.* describe how they implemented their own tables that could be queried in parallel using MapReduce. This is a much more complex and demanding approach than using off-the-shelf databases, which are built for the express purpose of querying and keeping indexes and data in sync. In the absence of a query planner,

all queries must be written as parallel MapReduce programs. Their approach does not tackle type-3 clones.

The idea of using bad hashes for clone detection was proposed by Baxter *et al.* [9]. Their approach creates bad hashes for sub-trees of the ASTs of classes, and thus requires full parsing of the source code in question.

Keivanloo *et al.* [1] show that the index-based approach scales to entire ecosystems. They build up a database of hashes for 18,000 Java programs. Their hashes are created for 3 consecutive lines while our hashes are created based on tokens. As a result we can detect type 3 clones that are generated by removing, adding, or changing a single token whereas their approach requires that three lines are exactly the same. Further, they use their own storage of the index, whereas we use an off-the-shelf database, Postgresql. This enables us to run elaborate queries, like "how much cloning exists between *different* projects" within hours, even without the use of parallelization.

III. Lightweight Approaches

To handle large amounts of data, we took Broder's [10] similarity metric and modified it towards greater speed. Instead of a distance metric, we compute bad hashes of the source code of each method. We compute three hashes: one for type-1 clone detection, another for type-2 clones, and one for type-3 clones. Detecting code duplication in an index is fast, because it does not involve cluster editing—and allows us to do without approximating algorithms.

To allow our algorithms to work independent of the programming language we use working definitions which slightly differ from the canonical ones. We feel this deviation is permissible since the canonical definitions emerge solely from ontological reasoning, rather than from empirical evidence.

A. Type 1: Hashes of source code

Type-1 clones are defined as "identical code fragments except for variations in whitespace, layout and comments" [11]. However, identifying comments is language specific and requires full parsing of the source code in most instances. In our approach, we do not ignore comments for this reason. As we show later in section §IV-B, even including comments, we detect a large number of type-1 clones across repositories. In our work we define type-1 clones as such:

Working Definition: Two documents are type-1 clones *iff* they differ in nothing but white-space.

The charm of this definition is that to find type-1 clones, it is enough to tokenize the input using the regular expression `/s+/`, concatenate the resulting tokens delimited by a separator, and then compute the SHA1 hash of the resulting string. Then, two snippets are type-1 clones *iff* they produce the same hash.

B. Type 2: Hashes of source code with renames

Type-2 clones are defined as "syntactically identical fragments except for variations in identifiers, literals, types, whitespace, layout and comments" [11]. We use the following definition of type-2 clones, which should not be in conflict with the canonical one:

Working Definition: Two documents are type-2 clones *iff* they are type-1 clones after every sequence of alphabetical letters is replaced by the letter "t", and all sequences of digits are replaced with the number "1". For an example, see Table I.

Table I – Example normalization of type-2 clones.

Source	Normalized

`myGetProviderFor: aSymbol`
`  |bound|`
`  bound := bindings at: aSymbol`
`    ifAbsent: [^nil].`
`  self assert: bound notNil.`
`  ^ bound`
	`t: t |t| t := t`
	`t: t t: [^t]. t`
	`t: t t. ^ t`

This is the same definition that has been successfully employed in detecting plagiarism [12] and it is computationally inexpensive. While this definition appears to be inclusive, as we will see in Table III, it catches barely more clones than there are type-1 clones.

C. Type 3: Shingles

Type-3 clones are defined as "Copied fragments with further modifications such as changed, added or removed statements, in addition to variations in identifiers, literals, types, whitespace, layout and comments" [11]. This definition leaves open just how much "further modification" is tolerable; clearly, it appeals to the intuitive sense of similarity. Broder [10] reports that defining *resemblance* based on shingles matches the intuitive sense of similarity in examining their data. We use this shingles-resemblance, which works as follows:

Let a "shingle" be a consecutive sequence of w tokens in a document, after the document has been transformed according to the rules of type-2 clones. The "sketch" of a document is a subset of its shingles.

Working Definition: Two documents are type-3 clones if and only if they share the same sketch.

By selecting only a subset of all the shingles two methods can be detected as similar even if they do not share all shingles. Also, their shingles do not need to appear in the same order to be detected as similar. While the selection should be random so as to not favor certain shingles over others, a document should also be equal to itself. Selecting shingles based on the bit representation of their hashes achieves just that.

In our implementation, shingles are sequences of four tokens (w=4), the hashes for the shingles are computed with SHA-1, and the subset constituting the sketch contains only those shingles whose hashes binary representation ends in "11". This selects an expected quarter of all shingles, since the digits of the binary representation of a hash each have an independent chance of $1/2$ to be '1'. Table II presents an example.

Note that it is not necessary to keep the shingles that make up the sketch. Rather, we can XOR them into one hash, which is a measure for whether or not two sketches are equivalent. This allows us to compute whether or not two documents are type-3 clones by checking whether their hashes are equal. Since clones that are too short are meaningless, we consider only documents that are at least 16 tokens long in our implementation.

Table II – Example normalization of type 3 clones. The underlined shingles are selected, because their binary representation ends in '11'. We only show the last 4 hex digits of hashes.

<table>
<tr><th>Normalized</th><th>Shingles</th><th>hashes</th></tr>
<tr><td>t: t |t| t
:= t t: t
t: [^t]. t
t: t t. ^
t</td><td>t: t |t| t, t |t| t :=,
|t| t := t, t := t t:,
:= t t: t, t t: t t:,
t: t t: [^]., t t: [^].
t, t: [^]. t t:, [^]. t
t: t, t t: t t, t: t t.
^, t t. ^ t</td><td>bd2d, c80b,
a3f8, 11b5,
6951, 4f55,
a43b, 8f58,
f7d2, d549,
bcee, fbe7,
84f4</td></tr>
</table>

While our definition of a type-3 clone is equivalent to Broder's Option B predictor with parameters $w = 4, m = 4$ [10, Theorem 1], it works out differently. Choosing only hashes that end in a certain bit pattern is proposed by Broder in an attempt to estimate the true resemblance, for which it is an unbiased estimate. Thus, he selects a subset of all shingles to improve performance and not, like us, to allow for deviation between similar code snippets.

IV. Empirical Study: Squeaksource

We used our approach to detect code duplication across repositories on Squeaksource (http://www.squeaksource.com); Squeaksource is the de-facto standard code repository in the Smalltalk ecosystem. As of June 2011, Squeaksource contains 2705 projects created by 3188 contributors over 7 years.

Each Squeaksource project is an individual repository. The version control system Squeaksource uses, called Monticello, creates a snapshot of the program (or package, depending on coding conventions) at every commit. The snapshot contains all of the program source code in a zipped text file, as a sequence of method definitions; this makes the method the natural granularity for our approach. Squeaksource amounts to a grand total of 47 GB of uncompressed data.

Projects in Squeaksource often include complete duplications of packages from other projects they depend on in their own repositories. The duplicated package has the same name as the original. This happens whenever a developer marks his own repository to depend on another repository. However, once stored, one cannot distinguish anymore between packages that directly belong to a project and those that come from the outside. Whether this inclusion of dependencies qualifies for code duplication or not may well be discussed. However, measuring it would report on the workings of Monticello more than on the behavior of developers. Therefore, while we stored all packages, regardless of origin, we tweaked our analysis to consider two methods to be cloned only if they were found both in different projects, and in differently named packages.

We compute and store all hashes of all versions of all methods and classes published on SqueakSource. We obtain a table in which each every hash is stored together with the clone-type it represents, and the places where it was found (a place is a tuple consisting of project, version, class, and method).

Table III – Percentage of cloned methods and classes out of 560,842 methods and 74,026 classes on SqueakSource.

	Type 1	type-2	Type 3
Percentage of cloned methods	14.55 %	16.33 %	17.85 %
Percentage of cloned classes	0.16 %	0.19 %	0.21 %

A. Space and time performance

We read a total of 22,641,865 method strings, which boil down to 560,842 different methods and 74,026 classes. For our purposes, similar to how Monticello stores class definitions, a class is merely the set of its methods, thus ignoring the inheritance hierarchy. For each method string, as well as for every class, we compute three hashes, one for each clone type. The data weighs in at merely 3.2 GB. However, due to alignment issues, they take significantly more space in memory. We store all hashes and method descriptors in a Postgresql database, where the data occupies 20 GB of space.[1]

Computing and storing all the hashes for the three techniques took 4:45 hours for all of Squeaksource (47 GB), on an 8 core Xeon at 2.3 GHz with 16 GB of RAM, using the Ruby 1.9.1 interpreter. Creating database indexes for every column took another 3 hours in total. Detecting code duplication across all projects then took only 2 hours. However, this also counted code duplication caused by the automated copying of Monticello, rather than willful code duplication. Removing these uninteresting clones was done with a database query that took another 10 hours of computation time. In contrast, the D-CCFinder experiment ran a single clone detection technique on 7.5GB of data for 51 hours, on 80 machines [5].

Since on more than twice the amount of data, and on ten times fewer cores, all three techniques together ran seven times faster, we can conclude that our lightweight approaches kept their promises regarding scalability.

B. Clones in the Squeaksource ecosystem

Table III shows the percentage of all methods across all versions and projects that were cloned in another project. We see that regardless of type, at least 14.5 % of all methods are present in at least two distinct repositories. Classes are cloned less frequently: only 0.16 %, or 115 classes in an entire repository, of all classes of all versions are straight copies from another package in another repository.

The table presents only a small increase in prevalence from type-2 clones to type-3. This shows that our definition of type-3 clones is rather restrictive. The reason for this is the following: if any one token changes, or is removed or added, then at most 4 shingles are removed from the document, and at most 4 are added. The chance of each shingle's hash to be part of the sketch is $1/4$. If none of the 4 removed shingles and none of the 4 added shingles is part of the sketch, then the sketch does not change. The chance of that happening is $(3/4)^8 \approx .1$. This is somewhat balanced by the fact that the 4 added and removed shingles don't have to be different, and that at the start and end of a document, changes involve fewer shingles. The

[1] The database can be accessed here: http://scg.unibe.ch/research/hot-clones.

high chance of the sketch changing explains why our working definition of type-3 clones in section §III-C clones is much more restrictive than it appears.

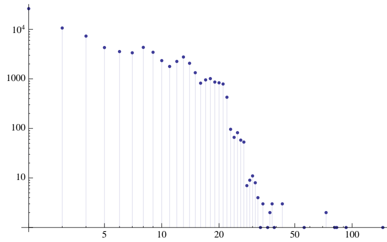

Figure 1 – Distribution of clone group sizes for type 1 clones. The x-axis is the size of the clone groups; the y-axis is the number of clone groups of that size across the ecosystem.

Figure 1 shows the distribution of type 1 clone groups according to their size. The distribution resembles a Pareto distribution. The median number of projects a cloned method is in is 3. There are large numbers of small clone groups, and few large clone groups. Note that some clone groups are very large, featuring hundreds of identical methods (in the case of type-1 clones). This is evidence that there are massive amounts of duplication in the ecosystem.

C. Multi-Version Analysis

We computed how many clones we would have missed using our approach, had we only looked at the latest versions of packages. Ignoring previous versions is plausible at first since code in repositories usually grow continuously. Furthermore, even if code changes after being cloned, type-3 clone detection might still find it. Setting aside the issue that determining the provenance of clones needs a version history anyways [3], this approach underestimates cloning by more than 20%.

We queried our data set for every method of which we know that it was cloned at some point in time, whether it is a type-3 clone in any latest version of any package. We found that when looking at the latest versions of all packages 24.4 % of all type-1 clones, 23.1 % of all type-2 clones, and 22.9 % of all type-3 clones would have been missed.

Note that more type-1 clones than type-2 clones are missed, and more type-2 clones than type-3 clones. Suppose that project A changes a method that was previously cloned by project B. Now, if we only look at latest versions, we may or may not detect this duplication as a type-3 clone. If, however, we look at all versions, we can detect the type-1 clone. Thus, more type-1 clones are missed than type-3 clones.

D. Threats to validity

In order to scale to larger amounts of data, we adopted slightly modified definitions of type 1, 2 and 3 clones. Thus the results may differ if more orthodox definitions of these clone groups are adopted. We have applied our techniques to a single ecosystem, which is comprised of Smalltalk systems only. Our findings may not generalize to other ecosystems, and other programming languages. Squeaksource is also considerably smaller than other ecosystems; if our techniques have been successful so far, it remains to be determined whether they scale to even larger code bases.

V. Conclusions

An index of bad hashes can detect type-1, type-2, and type-3 clones on large amounts of source code such as entire ecosystems—gigabytes of source code. Since bad hashing is such a cheap approach to clone detection, we can afford to index all versions, and thus detect clones that would otherwise be missed. In Squeaksource, 22.9 % of all type-3 clones are missed if only the latest versions of all packages are examined.

We found evidence for large amounts of duplication in the Squeaksource ecosystem. More than 14 % of all methods are copied from another package in another project. Regardless of one's opinion of code duplication: it is common.

Even though classes are meant to be modular, we have found that methods are reused in new contexts far more frequently than classes. Since projects on Squeaksource tend to stand for themselves, this suggests to us that this number is a good estimator of true duplication of source code, being used in different contexts to different ends.

Future work. Whether our definition of type-1, type-2 and type-3 clones is better or worse than the conventional one is yet to be determined. So far, evidence for the usefulness of clone type definitions is purely anecdotal. We will evaluate our definition agains Bellon's benchmark [11]. Furthermore, we plan to put our techniques to the test by applying them to other large ecosystems such as the Maven repository.

References

[1] I. Keivanloo, J. Rilling, and P. Charland, "Internet-scale real-time code clone search via multi-level indexing," in *WCRE*, 2011, pp. 23–27.

[2] C. K. Roy, J. R. Cordy, and R. Koschke, "Comparison and evaluation of code clone detection techniques and tools: A qualitative approach," *Sci. Comput. Program.*, vol. 74, pp. 470–495, May 2009.

[3] M. Kim, V. Sazawal, D. Notkin, and G. Murphy, "An empirical study of code clone genealogies," *SIGSOFT Softw. Eng. Notes*, vol. 30, no. 5, pp. 187–196, September 2005.

[4] T. Kamiya, S. Kusumoto, and K. Inoue, "Ccfinder: A multilinguistic token-based code clone detection system for large scale source code," *IEEE Trans. Software Eng.*, vol. 28, no. 7, pp. 654–670, 2002.

[5] S. Livieri, Y. Higo, M. Matsushita, and K. Inoue, "Very-large scale code clone analysis and visualization of open source programs using distributed ccfinder: D-ccfinder," in *ICSE*, 2007, pp. 106–115.

[6] M. S. Uddin, C. K. Roy, K. A. Schneider, and A. Hindle, "On the effectiveness of simhash for detecting near-miss clones in large scale software systems," in *WCRE*, 2011, pp. 13–22.

[7] J. Krinke, N. Gold, Y. Jia, and D. Binkley, "Cloning and copying between gnome projects," in *MSR*, 2010, pp. 98–101.

[8] B. Hummel, E. Juergens, L. Heinemann, and M. Conradt, "Index-based code clone detection: incremental, distributed, scalable," in *ICSM*, 2010, pp. 1–9.

[9] I. D. Baxter, A. Yahin, L. Moura, M. Sant'Anna, and L. Bier, "Clone detection using abstract syntax trees," in *ICSM*, 1998, pp. 368–377.

[10] A. Z. Broder, "On the resemblance and containment of documents," Jun. 1997, pp. 21–29.

[11] S. Bellon, R. Koschke, G. Antoniol, J. Krinke, and E. Merlo, "Comparison and evaluation of clone detection tools," *IEEE Transactions on Software Engineering*, vol. 33, no. 9, pp. 577–591, Sep. 2007.

[12] R. Brixtel, M. Fontaine, B. Lesner, C. Bazin, and R. Robbes, "Language-Independent clone detection applied to plagiarism detection," in *SCAM*, 2010, pp. 77–86.