# Clone detection that scales

Master Thesis

Simon Vogt

January 2014

**Thesis supervisors**:

Prof. Dr. Oscar Nierstrasz
and
Niko Schwarz
Software Composition Group, University of Bern

# Acknowledgements

First of all I'd like to thank Niko Schwarz for his encouragement and sharing his knowledge with me. I enjoyed the time spent at the SCG. Thank you Prof. Dr. Oscar Nierstrasz for supporting me and having me in your group. Thanks to my family and friends for your support and patience.

# Abstract

We show a clone detector that can scale to all source code ever written, in all versions. This is achieved by completely avoiding random reads from a central table, in other words all mappers and all reducers use, as input, only a contiguous slice of their input tables. Furthermore, we show the results of using our clone detector on 15,180 projects, all downloadable Java projects listed on the public meta repository "ohloh". On just three machines, we can finish clone detection across projects and versions in less than 20 hours. Besides offering a scaleable clone detector, our libraries allow anyone to download massive amounts of source code, store it space-efficiently and run cluster-style analysis on it, on commodity hardware. We intuit a reasonable configuration for the work load we have, and then systematically test the individual settings for performance, to tweak the overall configuration.

# Table of Contents

# 1 Introduction

In this thesis, we describe a clone detector that is conceptually similar to conQAT [7] (and should find roughly the same clones), but can compute the clones without the requirement of global memory. Without global memory, implementing the clone detection as a MapReduce job becomes easy, and leads to remarkable speed.

The asymptotic complexity of our clone detector is log-linear in the size of the input, as can easily be verified below for each step individually. The logarithmic factor stems from sorting the output of every stage before feeding it into the next.

The algorithm we describe is one long pipeline. We can summarize it as follows. We start by mining the Internet for software repositories. We download those repositories to local disk. We convert the repositories to git format. We copy the repositories into a distributed file system. We unpack the repositories into Bigtables. We run a sliding window of 5 lines over each source file. If we find a similar snippet in another file, we have found a clone and try to expand the clone to maximum size. Once all clones are found, we run a filter over the results to increase precision.

Altogether, the entire pipeline runs in 3 computers in under 48 hours, including downloading the repositories. Clone detection itself takes 19 hours, 30 minutes, for 15,180 Java projects and all tagged versions in each. Our cluster setup consists of three server machines, with 8, 8 and 32 cores, and 16, 16 and 120 GB RAM, respectively.

We detect clones in 15,180 projects, including all commonly heard of Java frameworks and libraries, in all tagged versions. We detect clones across 16,953,815 different Java methods, totaling 476,069,312 lines. In this metric, we count Java methods only once, even if the same method appears verbatim in a different file, or in a different project. The compressed and fully packed git repositories total 105 GB of data. The Snappy-Compressed tables (using HBase[1]) containing all versions, projects, functions and snippets total 53.9 GB. We will discuss the remarkable efficiency of our storage.

We simplify the management of a heterogenous cluster with a distribution mechanism. We show how we tune the cluster for our specific MapReduce jobs and how we benchmark different configurations.

[1] `http://hbase.apache.org`

# 2

# Clone detection using bad hashes in a nutshell

At its core, our approach uses 'bad hashes', *i.e.* hashes that may collide, to define the similarity of two snippets. When two snippets of code produce the same bad hash, they are considered similar.

Kamiya *et al.* proposed the following rules to abstract minutiae irrelevant to overall similarity [8]. We apply all of them. (RJ1) Remove package names, (RJ2) Supplement callees, (RJ3) Remove initialization lists, (RJ4) Separate class definitions, (RJ5) Remove accessibility keywords, (RJ6) Convert to compound block.

Let a 'snippet' be a run of five consecutive lines of code. Let us recap.

- Two snippets are detected as type-1 clones *iff* they differ in nothing but white-space, after all rules (RJ1) through (RJ6) are applied.
- Two snippets are detected as 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 2.1.
- Two documents are detected as type-3 clones if and only if they share the same sketch.

The last definition hinges on the definition of a *sketch* of a snippet, which we define as follows: Let $s$ be the set of all (possibly overlapping) runs of four consecutive tokens of a snippet. Now compute the SHA1 hash of all elements of $s$ and discard all elements whose hash's binary representation does not end in "11". The set of remaining elements of $s$ is the hash. This definition selects roughly a quarter of all the elements of $s$, since the digits of the binary representation of a hash each have an independent chance of 1/2 to be '1'.

**Tab.** 2.1.: Example normalization of type-2 clones.

| Source | Normalized |
|---|---|
| `myGetProviderFor: aSymbol \|bound\|`<br>`  bound := bindings`<br>`at: aSymbol ifAbsent:`<br>`  [^nil]. self`<br>`assert: bound notNil.`<br>`  ^ bound` | `t: t \|t\| t := t`<br>`t: t t: [^t]. t`<br>`t: t t. ^ t` |

Note that, for all three clone types, to test whether or not two snippets are similar, we never have to compare the snippets themselves. All we have to do is abstract both snippets, compute the SHA1 hashes of the abstractions, and compare the hashes.

This somewhat involved definition of type-3 clones pays back manyfold in performance, while achieving good precision. Let us discuss these two claims in reverse order. As for precision, it suffices to see that there are only marginally more type-3 clones than there are type-1 clones [15]. However, the precision of type-1 clones is, for sufficiently large clones, 100 %. Since there are, as a percentage, only few more type-3 clones than type-1 clones, precision must be high. As for performance, our definitions for type-1, type-2, and type-3 clones form equivalence relationships. Clustering equivalence relationships is trivial. In our case, we must only sort all snippets by their hashes, and then each cluster will be at consecutive positions after sorting. However, sorting across clusters is efficient and very well understood.

As an example, let us consider detecting clones in the three functions in Figure 2.1. We move a sliding window of 5 lines over every source file. For each snippet, we compute three normalizations, one for each clone type, and from the normalization, the resulting *bad hash*. The first five normalized snippets and their hashes are seen in Figure 2.2.

This process of subsequent normalizing and hashing will produce a sequence of hashes for every input function. From this point on, the raw source files are no longer needed and we can detect clones using the sequences of hashes alone. If we now write the snippet into a sorted table, under the row key of its *bad hash*, together with location data of the snippet, then we will find all colliding snippets as a consecutive subtable. As we will see below, with only a few efficient transformations, we can achieve a slightly modified table where we can find all collisions for a function in order. As seen in Figure 2.1, functions `FUN1` and `FUN2` collide in three different snippets. Since the colliding snippets between `FUN1` and `FUN3` are less than 10 lines apart, we merge all of them, and the lines between them, into one *clone*. Merging all mergeable collisions between pairs `FUN1`–`FUN2`, `FUN1`–`FUN3`, `FUN2`–`FUN3` produces, as seen Figure 2.2, three clones. They are the output of the clone detector.

Finally, we filter out all clones that are less than 10 lines long. Then, for every clone, we compute the set of identifiers on both sides of the clone. If the intersection of their identifiers is smaller than 85 % of the size of their union, we discard the clone. The latter rule has been proposed and validated by Koschke [10]. It follows from the intuition that if code is copied, the invoked API methods cannot be renamed and aggressive renaming of identifiers is unusual.

FUN1

```
             public static int log10(int x, RoundingMode mode) {
               checkPositive("x", x);
               int logFloor = log10Floor(x);
               int floorPow = POWERS_OF_10[logFloor];
               int result = -1;
FUN2:          switch (mode) {
               case UNNECESSARY:
                    checkRoundingUnnecessary(x == floorPow);
                    // fall through
               case DOWN:
                    result = logFloor;
               case CEILING:
               case UP:
FUN2,FUN3:          result = (x == floorPow) ? logFloor : logFloor - 1;
FUN2,FUN3:     case HALF_DOWN:
               case HALF_UP:
               case HALF_EVEN:
                    // sqrt(10) is irrational, so log10(x) - logFloor is never exactly
                    // 0.5
                    result = (x <= HALF_POWERS_OF_10[logFloor]) ? logFloor : logFloor - 1;
               }
               return result;
             }
```

FUN2

```
FUN3:        public static int log10(int x, RoundingMode mode) {
               int logFloor = log10Floor(x);
               int floorPow = powers_Of_10[logFloor];
FUN1:          switch (mode) {
               case UNNECESSARY:
                    checkRoundingUnnecessary(x == floorPow);
                    // fall through
FUN3:          case FLOOR:
FUN3:          case DOWN:
FUN3:               return logFloor;
               case CEILING:
FUN3:          case UP:
FUN1,FUN3:          return (x == floorPow) ? logFloor : logFloor + 1;
FUN1,FUN3:     case HALF_DOWN:
FUN3:          case HALF_UP:
FUN3:          case HALF_EVEN:
FUN3:               // sqrt(10) is irrational, so log10(x) - logFloor is never exactly
FUN3:               // 0.5
                    return (x <= half_Powers_Of_10[logFloor]) ? logFloor : logFloor + 1;
               default:
                    throw new AssertionError();
               }
             }
```

FUN3

```
FUN2:        public static int log10(int x, RoundingMode mode) {
               int logFloor = log10Floor(x);
               int floorPow = powers_Of_10[logFloor];
               switch (mode) {
FUN2:          case FLOOR:
FUN2:          case DOWN:
FUN2:               return logFloor;
               case CEILING:
FUN2:          case UP:
FUN1,FUN2:          return (x == floorPow) ? logFloor : logFloor + 1;
FUN1,FUN2:     case HALF_DOWN:
FUN2:          case HALF_UP:
FUN2:          case HALF_EVEN:
FUN2:               // sqrt(10) is irrational, so log10(x) - logFloor is never exactly
FUN2:               // 0.5
               return (x <= half_Powers_Of_10[logFloor]) ? logFloor : logFloor + 1;
               default:
                    throw new AssertionError();
               }
             }
```

**Fig.** 2.1.: Collisions of similar snippets between three functions. The collisions for each snippet are given on the left, in bold. `FUN2` in bold means that the following five lines (a snippet) collide with a snippet from `FUN2`. `FUN1` and `FUN2` collide in three snippets, `FUN1` and `FUN3` collide in 2 snippets, `FUN2` and `FUN3` collide in 11 snippets. The frames show the merged snippet collisions, which we call *clones*.

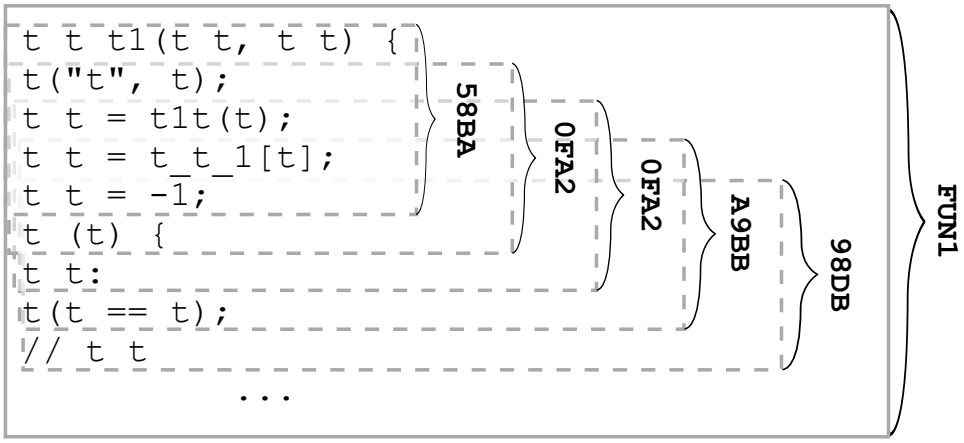


**Fig.** 2.2.: The first few snippets and their hashes, for the first function in Figure 2.1.

# 3 Pipeline



Even though our approach can be thought of as just one long pipeline, it can be broken into two parts. The first part is concerned with moving data into the distributed file system; the second part reads from the distributed file system and writes clone data into it. The first part is implemented in Ruby, and concurrency is achieved using GNU parallel [16] and Unix pipes. The second part is implemented in Java and achieves parallelism using a MapReduce pipeline.

## 3.1. Mining the Internet for source code

In this section, each subsection corresponds to a separate Ruby script in our pipeline. The scripts are connected via Unix pipes, so that the output of one script forms the input of the next, leading to a naturally concurrent implementation, without our having to worry about locks.

We further increase parallelism by running stages of the pipeline using GNU parallel [16]. This is powerful, as it even allows distributing the computation seamlessly across our cluster, using the `-sshloginfile` switch. It is also a very clean design: even though,

conceptually, our computation is parallel, concurrent and even distributed, it makes little difference for understanding the individual steps of the pipeline.

Ohloh aims to be a nearly comprehensive online directory of open source projects, regardless of where they are hosted. By June 2013, it had nearly 600,000 projects listed.

### 3.1.1. Produce a list of all project names

We crawl the ohloh website to produce a list of Java project names. The URL `https://www.ohloh.net/p?q=java` returns a list of 50,000 projects spread over 5,000 html pages. We parse all html pages to extract the project names of each project.

### 3.1.2. Gather download URLs

For each project name, we produce a list of its download URLs. Ohloh offers an http API that lists all download URLs, given a project name. For example, to get the list of all download URLs for project Tomcat, `http://www.ohloh.net/p/tomcat/enlistments` produces all download URLs.

After downloading all enlistments, we found download URLs to 3077 Git repositories, 662 Mercurial repositories, 15982 Subversion repositories and 6418 CVS repositories. Note that there can be several download URLs, or none, for a project.

### 3.1.3. Choosing the best download URL

To choose the best download URL for a project, we use a hand-crafted scoring system. The download URL with the highest score gets sent to the next stage.

- We boost a repository score by 75 points if the URL contains one of the following keywords: src, source, core, standard, build, master. On the other hand we lower a repository score by 75 points if the URL contains one of the following keywords: doc, docs, extras, extra, tool, tools, test, testing.
- For each time time that the project name occurs in the download URL, the score gets increased by 10.
- We add $1000/l$, where $l$ is the length of the repository URL. This follows from the observation that the projects' official repository URL tends to be short.

### 3.1.4. Download repositories and convert to git format

For each download URL, we download the repository. If the project cannot be downloaded as a bare (*i.e.* without a checked out version) git repository, we convert it into one. In the case of Subversion repositories, we do not use the git-svn scripts from the git distribution, but instead check out the repository using svn directly, and commit the contents into a new git repository.

The output of this stage is, unlike all previous stages, not written to stdout, but is a directory on the local disk containing all repositories in git format.

### 3.1.5. Move local repositories into distributed file system

```
# pack-refs with: peeled
1f8d1c6b1d8d refs/heads/master
bf2e04ae5fbe refs/tags/v0.1
```

**Fig.** 3.1.: Example of a pack-refs file. Each line represents a version of, named by the right-hand side. So, version v0.1. is found in the commit of SHA1 hash bf2e04ae5fbe.

In a bare git repository that's fully packed (this is ensured by 'git repack'), two files suffice to describe all versions: the packfile and the pack-refs file. The pack-refs file contains all named versions (or 'tags' in git parlance), the packfile contains the source trees and some meta information. See Figure 3.1 for an example of a pack-refs file.

For each repository, we move these two files into the distributed file system. Fully packed repositories are more simply structured than general repositories, and use less space.

The median size of a pack file is 1.5 megabytes. Hadoop's distributed file system, HDFS, effectively sports a minimum file size of 64 megabytes, meaning that a naive import would cost more than an order of magnitude in space consumption. Hadoop offers an archive format similar to tar files, called HAR files. Before creating the HAR file we generate an index file holding the path and size of all contained pack files. Running a 'find' command inside HDFS on the HAR file otherwise is tremendously slow. After moving all pack files and pack-ref files into one big HAR file, space is efficiently used on the distributed file system. The HAR file takes up only 103.1 GB of space on the distributed file system.

## 3.2. MapReduce pipeline

Let us now describe the second half of the pipeline, where all projects have already been downloaded and now reside in the distributed file system. This is the algorithmic part of the pipeline. Note that at no point is there a requirement to read from arbitrary tables. We use *Cells* [14] as our MapReduce pipeline. It doesn't distinguish between mappers and reducers, and therefore, neither will we. Every step can be implemented by mappers that accept only a slice of the overall input data, and produce their own slice of the overall output data, without ever needing more than their own slice. An overview of the pipeline is seen in Figure 3.2. Let us discuss all mappers in turn.

### 3.2.1. Populate

In this mapper, we merge all incoming git repositories into one, split files into functions, and compute all hashes. Git stores a software repository in a persistent data structure [6], more specifically, as a directed acyclic graph of immutable nodes. Files are the leaves of this graph. The advantage of this model is that if the same file appears in two source trees, the leaf representing that file can be shared between the source trees. This idea is easily generalized to allow sharing of identical files across project repository boundaries, as exemplified in Figure 3.4.

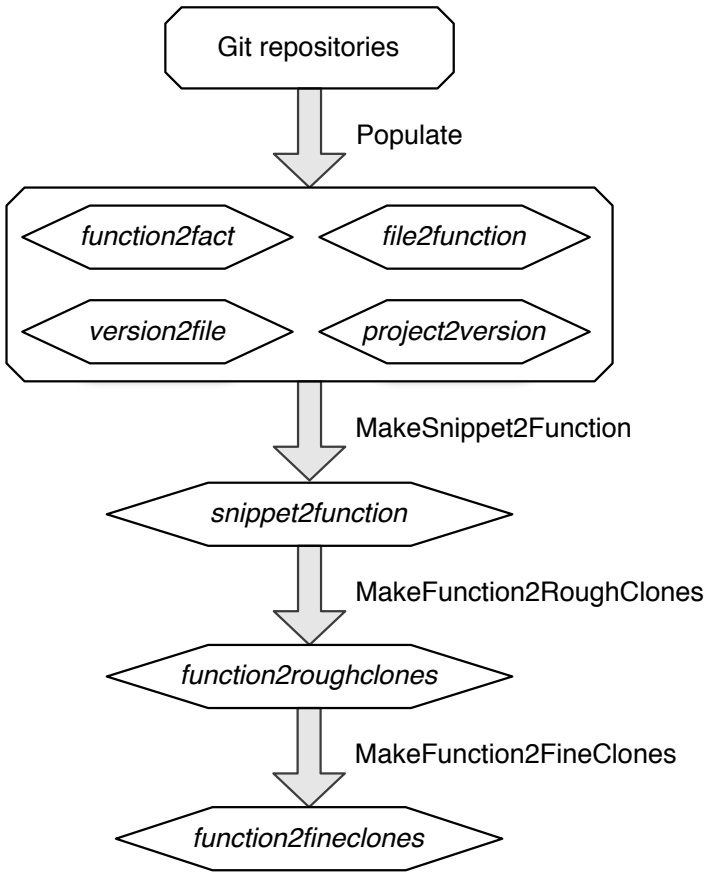


**Fig.** 3.2.: The pipeline architecture of our algorithm.

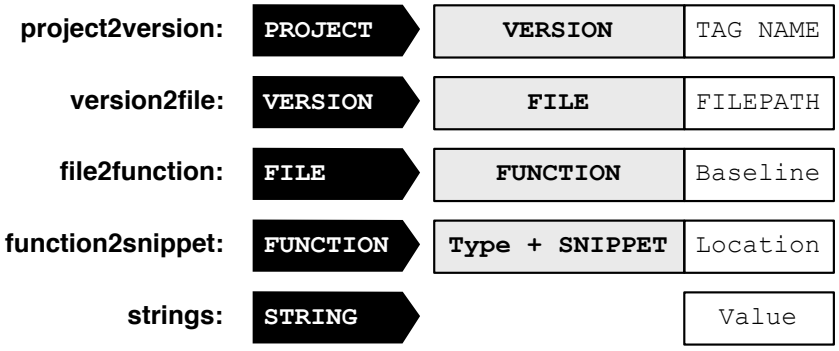


**Fig.** 3.3.: The table design for storing all repositories using the flyweight pattern. Names printed in all caps are hashes.

This mapper produces several tables whose layout can be seen in Figure 3.3. For every version in the input project, we walk the entirety of its source tree, and write the entire source tree into just one table row. Note that HTable poses no limit on the number of columns per row.

For each file we write one row to table 'version2files'. Its cell key is the SHA1 hash of the file contents. Its column key is the file name. The file contents itself is stored separately, in a 'strings' table, where we store a cell whose row key is the SHA1 hash of the file contents, and the cell value is the raw file contents. Thus, no matter how many projects share the same file, the file contents is stored exactly once, in the 'strings' table. This is illustrated in figure 3.4.

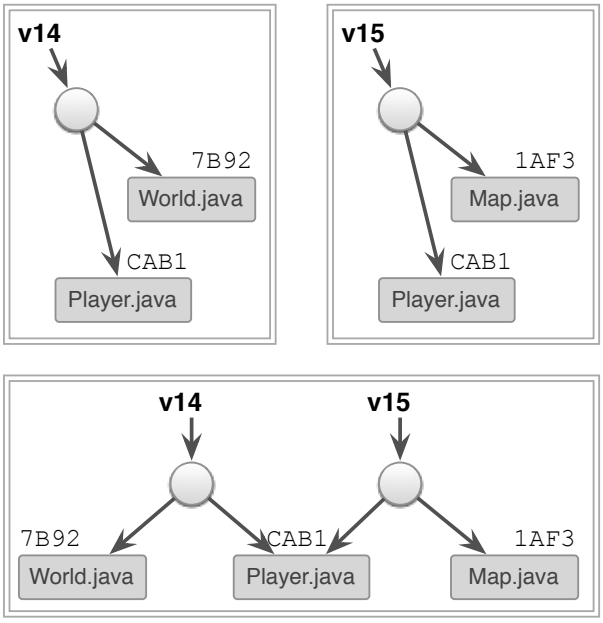


**Fig.** 3.4.: Versions v14 and v15 (possibly from different projects!) both contain a file named Player.java, whose contents has hash CAB1 (above). We save space by forcing v14 and v15 to share Player.java, thus preventing the duplication of Player.java in our database.

We use the same trick throughout: even if a function appears in multiple files, there is only one row in table 'function2snippet' for that function. Effectively, we're using the Flyweight pattern to reduce our data size, just as Git does.

To apply rules (RJ1) through (RJ6), we use lightweight parsing on Java source files. Our parsing has to be powerful enough to identify class definitions, function definitions, package names, method invocations, initialization lists, remove accessibility keywords, and the statements `if, while, for` with the following statement.

Using our regular expression engine [13], we can parse this in one pass, from one single regular expression. Since the resulting regular expression is quite involved, let us discuss a simpler one, identifying all functions and classes.

Class definitions are much easier to express as a regular expression, since Java has a reserved keyword for them. A class definition can be expressed as `\bclass\b.*?\{`.

Since Java's function declaration uses no keyword, functions are the hardest thing to parse from a regular expression alone. However, the following features of function definitions

can be exploited to craft a regular expression that works well in practice: there are at least two distinct words before an open parenthesis (method name and return type), there's exactly one open and one closing parenthesis, followed by exactly one opening curly brace. Semicolons, periods, etc, are disallowed.

Let `FUNC` be our regular expression to parse function declarations, and `CLASS` our regular expression to parse class declarations, then the following parses Java source files, splitting them into classes and functions: `.*?(CLASS.*?(FUNC.*?)*?)*?`.

Note that our regular expression contains nested asterisks. In a back-tracking implementation, this could easily lead to exponential run-time. However, our algorithm is guaranteed to be linear in the input size, even for a complicated regular expression like ours.

The other features are added to the regular expression analogously. Our regular expression engine will produce a full parse tree from this regular expression. Walking this tree, we can apply all rules at once.

The main output of this mapper is table 'function2snippet', the other tables are side outputs. This means that the next mapper will receive, as its input, table 'function2snippet'.

In our example, after loading in the three source files from Figure 2.1, the main output is seen in Table 3.1.

### 3.2.2. Reverse index

In this mapper, we build a reverse index from snippet hashes to the functions that contain them. This simply means to exchange row and column keys in Table 3.1. After this is done, snippets that have collisions will appear in the same row. For our data, the output of this stage is 21.8 GB in size, snappy-compressed.

### 3.2.3. Filter

In this mapper, we remove all rows from the previous stage that have at most one column. In other words, we ignore snippets that are found in at most one function, since they can *per definitionem* not be part of a clone, and treat popular snippets specially.

This stage reduces the amount of input data we deal with substantially. The output of this mapper is 5.2 GB in size, and another 294 MB in the side table of popular snippets.

### 3.2.4. Rough clones

In this mapper, we gather, for every function, all collisions with it. For example, in the example in Table 3.3, the first row contains all collisions that involve `FUN1`, ordered first by colliding function, and second by position of the collision in `FUN1`. The input to this mapper is a table that maps from a snippet to all functions containing it. As can be seen in the pseudo code of this operation in Figure 3.5, the output of this mapper is of size $O(c^2 n)$, where $n$ is the number of functions with collisions, and $c$ is the maximal number of functions that a snippet occurs in.

Since $c$ can reach substantial sizes, $c$'s distribution can be seen in Figure 3.6, we have to introduce special treatment to snippets that occur in very many functions, thus providing

**Tab.** 3.1.: Table that maps from functions to snippets.

| Function | Snippet | Position | Snippet | Position | Snippet | Position |
|---|---|---|---|---|---|---|
| FUN1 | 0FA2 | 2 | 1474 | 12 | 20A4 | 13 |
| | 2786 | 15 | 3017 | 8 | 38C3 | 6 |
| | 48C3 | 17 | 58BA | 0 | 5985 | 7 |
| | 5F72 | 10 | 6F45 | 14 | 98DB | 4 |
| | 9C62 | 11 | A4A8 | 9 | A9BB | 3 |
| | BB3E | 5 | | | | |
| FUN2 | 03D8 | 9 | 1FF0 | 2 | 20A4 | 12 |
| | 4FE1 | 11 | 54F4 | 4 | 6F45 | 13 |
| | 8889 | 1 | 9721 | 0 | A037 | 7 |
| | ABB0 | 15 | BB3E | 3 | BFEA | 14 |
| | C751 | 17 | D0AF | 16 | E444 | 6 |
| | ECBB | 5 | FC06 | 8 | | |
| FUN3 | 03D8 | 6 | 20A4 | 9 | 4FE1 | 8 |
| | 5752 | 3 | 6F45 | 10 | 9721 | 0 |
| | A037 | 4 | ABB0 | 12 | B437 | 2 |
| | BB95 | 1 | BFEA | 11 | C751 | 14 |
| | D0AF | 13 | FC06 | 5 | | |

**Tab.** 3.2.: Output of the reverse index mapper. Row keys are snippets, cell values are functions that contain these snippets.

| *Snippet* | *Function* | *Position* | | | | | |
|---|---|---|---|---|---|---|---|
| 03D8 | FUN2 | 9 | FUN3 | 6 | | | |
| 0FA2 | FUN1 | 1 | | | | | |
| 1474 | FUN1 | 12 | | | | | |
| 1FF0 | FUN2 | 2 | | | | | |
| 20A4 | FUN1 | 13 | FUN2 | 12 | FUN3 | 9 | |
| ... | | | | | | | |
| 4FE1 | FUN2 | 10 | FUN3 | 8 | | | |
| ... | | | | | | | |
| 6F45 | FUN1 | 14 | FUN2 | 13 | FUN3 | 10 | |
| ... | | | | | | | |
| 9721 | FUN2 | 0 | FUN3 | 0 | | | |
| ... | | | | | | | |
| A037 | FUN2 | 7 | FUN3 | 4 | | | |
| ... | | | | | | | |
| ABB0 | FUN2 | 15 | FUN3 | 12 | | | |
| ... | | | | | | | |
| BB3E | FUN1 | 5 | FUN2 | 3 | | | |
| ... | | | | | | | |
| BFEA | FUN2 | 14 | FUN3 | 11 | | | |
| C751 | FUN2 | 17 | FUN3 | 14 | | | |
| D0AF | FUN2 | 16 | FUN3 | 13 | | | |
| ... | | | | | | | |
| FC06 | FUN2 | 8 | FUN3 | 5 | | | |

```
Input: A map from snippets to all of its locations.
  A location is a line and a function.
Output: A map from function to all of its collisions.
  A collision is a pair of locations.

for entry : inputMap {
  for thisLocation : entry.locations {
    for thatLocation : entry.locations {
      if thisLocation == thatLocation {
        continue
      }
      cell := new Cell()
      cell.setRowKey(thisLocation.function)
      cell.setColumnKey(concat(
        thatFunction.function, thisLocation.line))
      write(cell)
    }
  }
}
```

**Fig.** 3.5.: The pseudo code that produces Table 3.3 from input Table 3.2

an upper bound for $c$. As seen in Figure 3.6, skipping snippets that occur in more than 1,000 functions, suppresses only 0.04 % of all snippets, but dramatically decreases the output size. In fact, only this restriction ensures that the output of this mapper is linear in its input.

Snippets that occur in more than 1000 functions, we still output, but into a side table named 'popularSnippets'. We still use these snippets in matching, but only if a snippet that occurs less frequently than some threshold[1] also matches. This serves as a useful filter: if a snippet is extremely common, it probably represents merely an artifact of the programming language or API, rather than code duplication worth studying.

In our example, for a threshold of 3, the output of this mapper in Tables 3.3 and 3.2.

**Tab.** 3.3.: The table from functions to collisions. For our data set, this table is 32.4 GB in size.

| *thisFunction* | *thatFunction, Position in thisFunction* | | *Snippet, Position in thatFunction* | | | |
|---|---|---|---|---|---|---|
| **FUN1** | **FUN2, 5** | BB3E, 3 | **FUN2, 13** | 20A4, 12 | **FUN2, 14** | 6F45, 13 |
| | **FUN3, 13** | 20A4, 9 | **FUN3, 14** | 6F45, 10 | | |
| **FUN2** | **FUN3, 0** | 9721, 0 | **FUN3, 7** | A037, 4 | **FUN3, 8** | FC06, 5 |
| | **FUN3, 9** | 03D8, 6 | **FUN3, 11** | 4FE1, 8 | **FUN3, 12** | 20A4, 9 |
| | **FUN3, 13** | 6F45, 10 | **FUN3, 14** | BFEA, 11 | **FUN3, 15** | ABB0, 12 |
| | **FUN3, 16** | D0AF, 13 | **FUN3, 17** | C751, 14 | | |

[1] In our experiments, threshold $c = 1000$ worked out well.

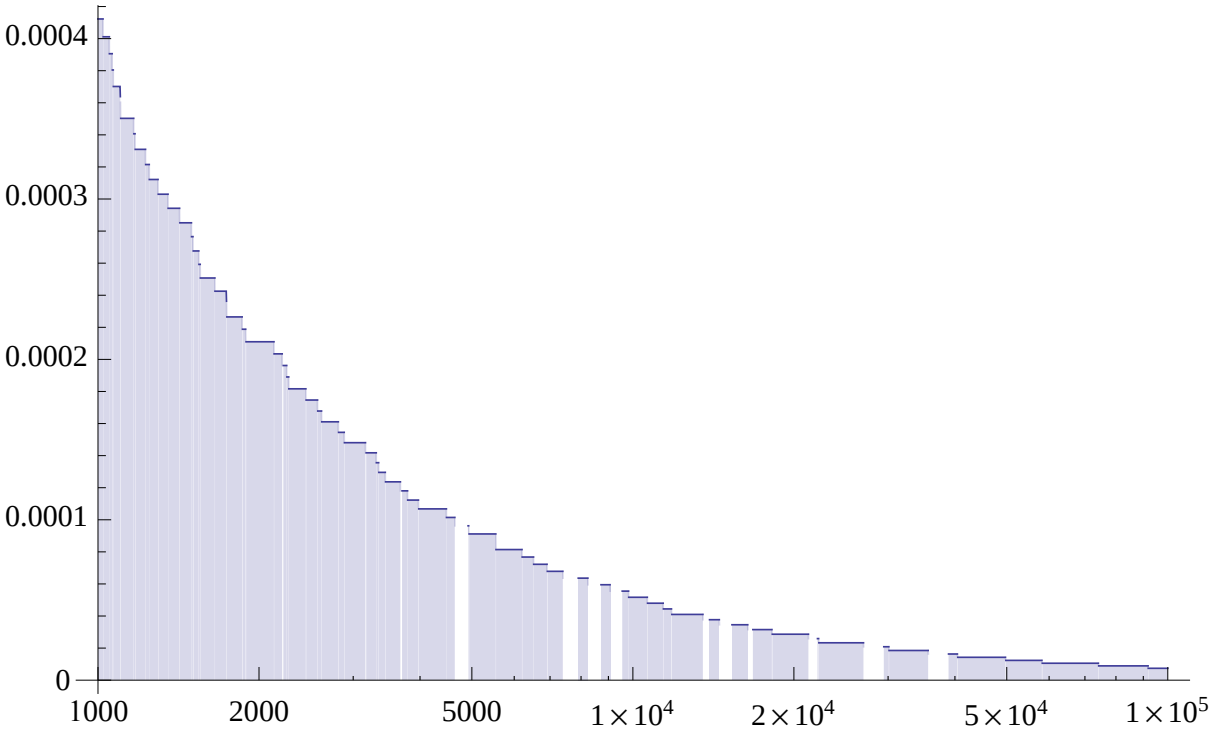


**Fig.** 3.6.: Percentage of snippets that have more than n clones. The $y$-axis shows the percentage, the $x$-axis shows the number of clones. If a snippet occurs in two places, it has two clones. The $x$-axis is shown in range $[1000, 100000]$.

**Tab.** 3.4.: The content of the popular snippets Table. For our data set, this table is 293.3 MB in size.

| *Function* | *Snippet* | *Position* | | |
|---|---|---|---|---|
| **FUN1** | **20A4** | 13 | **6F45** | 14 |
| **FUN2** | **20A4** | 12 | **6F45** | 13 |
| **FUN3** | **20A4** | 9 | **6F45** | 10 |

To simplify the code of this mapper, we assume that a snippet can occur at most once in a function. We tested this assumption. We sampled 405 Git projects with a total of 9,410,813 functions in normalized form. We count 210,774 functions where our assumption that one snippet occurs at most once in a function is violated. That is about 2 %.

### 3.2.5. Clone expansion

The input to this mapper is seen in Table 3.3: Rows contain one function in the row key, and the sorted list of matching snippets with another function, in the column key. The snippets are sorted with respect to their occurrence in the row key function, meaning they may be out of order with respect to the function in the column key.

The snippets are stitched together into clones. A clone is a match between two functions, and can be arbitrarily long. It is no longer restricted to 5 lines. When stitching together clones, we make sure that the gap in each clone, in either of the involved functions, is at most 4 lines long. If the gap is greater than that, more than one clone is output. While the popular snippets from table 3.4 are never used to initiate a clone, they are used during stitching, and may prevent a gap from becoming too large.

Note that our promise to avoid lookups into a global table was not violated by using the popular snippets table. The full size of the table is only 500 megabytes, meaning we can copy it in full to all mappers, avoiding a central lookup completely.

The output of this mapper is seen in table 3.5. Going back to the input files in figure 2.1, we see that we have detected, meaningful and intuitive clones for this input.

**Tab.** 3.5.: Final output, the detected clones.

| | **Clone 1** | | **Clone 2** | | **Clone 3** | |
|---|---|---|---|---|---|---|
| **Function** | FUN1 | FUN2 | FUN1 | FUN3 | FUN2 | FUN3 |
| **Position** | 5 | 3 | 13 | 9 | 0 | 0 |
| **Length** | 14 | 15 | 6 | 6 | 22 | 19 |

**Clone filter** For every clone, we compute the set of identifiers on both sides of the clone. If the intersection of their identifiers is smaller than 85 % of the size of their union, we discard the clone.

In our example, the vocabulary for all snippets is above the threshold, meaning that all survive filtering.

# 4
# Cluster Management



Hadoop does not provide a solution to configure different machines with different hardware configurations in a cluster easily. Due to the heterogeneity of our cluster we created our own distribution mechanism.

We had to deal with a cluster consisting of only three machines. The one with 128GB RAM was dedicated to this project and was solely used to run HBase and Hadoop on it. The other two machines were sporadically used by the SCG staff and running other minor jobs so there could be a performance drawback from time to time. They were also not that powerful (16GB RAM each). To install Hadoop and HBase on all machines we relied on CDH[1].

## 4.1. Cluster configuration

Hadoop has configuration options for Core-, HDFS-, MapReduce- and Yarn-specific settings. The default settings can be overwritten by using site-specific configuration files. These files (`{core|hdfs|mapred|yarn}-site.xml`) can be found and adjusted in the `/etc/hadoop/conf/` directory on each node. To distribute the configuration settings to the different nodes, we use a simple distribution mechanism.[2]

On the cluster machines, symlink the `{hdfs|mapred|yarn}-site.xml` configuration files to `core-site.xml`. Now, whenever we want to adjust settings, we refer to this one file.

In our repository, we maintain a file with global settings that all nodes share, and another set of files for node-specific settings, one for each node. Then, whenever we change a setting in our repository, we run a script to broadcast the change to all nodes. The script first merges the global with the node-specific settings, and then creates the `core-site.xml` and `hadoop-env.sh` files for each node. The generated files will then automatically be uploaded to the `/tmp/` folder of each node.

[1] Cloudera Distribution Including Apache Hadoop: `http://www.cloudera.com/content/cloudera/en/products-and-services/cdh.html`

[2] Note that we applied exactly the same procedure to simplify the configuration of the `{hadoop|yarn}-env.sh` as well as of the whole HBase configuration files.

```
simon@lm:~/clone-detector$ vim hadoop-conf/leela/node-properties.xml
simon@lm:~/clone-detector$ ./scripts/cluster/upload_configurations.sh leela
Successfully uploaded configuration files to leela.
Please run ‘sudo /tmp/node_update_hadoop.sh‘ or ‘sudo /tmp/node_update_hbase.sh‘ in
    csshX.
```

**List.** 4.1: Updating a node's configuration

The following prerequisites are required:

**Machines** All nodes need to be listed line-separated in a file called "machines".

**Login** The login name needs to be stored in a file called "sshuser". The file will be used by the `rsync` command to upload the configuration file to the node with the specified user. SSH-password-less login should be set-up beforehand.

**Cluster SSH tool** csshX[3] is a terminal prompt which forwards the input typed to all remote machines. It needs to be installed on the master machine.

After running `upload_hadoop_configurations.sh` csshX automatically opens terminal windows to all nodes. Normally, we want to copy the generated configuration file from `/tmp/` to `/etc/hadoop/conf/` and restart the Hadoop services.
The script `node_update_hadoop.sh` which was also copied automatically by the upload script will do exactly that for us. So we will simply execute the command `sudo ./tmp/node_update_hadoop.sh` in the master window of csshX.

[3]csshX: `https://code.google.com/p/csshx/`

# 5 MapReduce tuning

We show how to tune a Hadoop job on the example of our populate step (see section §3.2.1). The population stage generates data from a relatively small input, namely the source files of the repositories. The input to this job is significantly smaller than its output.

## 5.1. Map Input

What we effectively have as input are Git pack files. JGit is used to parse the pack files. By default, the whole job gets marked as failed if one of its mappers fails. We tackle this by catching all exceptions, logging them and incrementing counters to get real-time updates during execution. So the job never fails - even if a corrupt repository gets processed. Therefore, we also adjust the default allowed map failure number for a successful job from 0 to 100. Now we can safely ignore failed map attempts regarding the job aliveness.

```
1 jobConf.setInt(MRJobConfig.MAP_FAILURES_MAX_PERCENT, 100);
```

**List.** 5.1: Property mapreduce.map.failures.maxpercent

## 5.2. Running the Map Tasks

Hadoop can run multiple instances of a map task in parallel if it has the impression that a map job runs longer than it usually does. This is called speculative execution and should avoid long running tasks on slow nodes to rate-limit the whole job. In our environment, this does not make much sense since there are no nodes in our cluster which have serious performance drawbacks so that it would be worthwhile to spawn a backup task on a second node. Therefore we turned speculative execution off.

```
1 jobConf.setBoolean(MRJobConfig.MAP_SPECULATIVE, false);
```

**List.** 5.2: Property mapreduce.map.speculative

By default Hadoop, allocates 1536 MB of memory for each map task[1]. Depending on the repository size and complexity, we get OutOfMemory exceptions while we extract the source files with jGit. We solved this by increasing the reserved memory per task.

```
1 jobConf.setLong(MRJobConfig.MAP_MEMORY_MB, 4000);
2 jobConf.set(MRJobConfig.MAP_JAVA_OPTS, "-Xmx3500m");
```

**List.** 5.3: Property mapreduce.map.memory.mb and mapreduce.map.java.opts

## 5.3. Map Output

Depending on the size of the processed source files of a pack file, we get different output in size. Hadoop spills the output to disk whenever it finishes or reaches a certain threshold. It is best, when spilling happens exactly once per map task [18, p. 211]. With Hadoop's default settings, we observed several spills to disk. We had to adjust three values that seemed to have a positive impact on our processing.

```
1 jobConf.setInt(MRJobConfig.IO_SORT_MB, 500);
```

**List.** 5.4: Property mapreduce.task.io.sort.mb

```
1 jobConf.setInt(MRJobConfig.IO_SORT_FACTOR, 50);
```

**List.** 5.5: Property mapreduce.task.io.sort.factor

```
1 jobConf.setFloat(MRJobConfig.MAP_SORT_SPILL_PERCENT, 0.9f);
```

**List.** 5.6: Property mapreduce.map.sort.spill.percent

The additional spilling of map output seriously affected the performance of the jobs. We increased the default value from 100 to 500. This gives the memory buffer bigger capacity while sorting the map output. As we increased the `mapreduce.task.io.sort.mb` we also increase the sort factor to 50. We increased the threshold of the serialization buffer for spilling map output to 90% instead of the default of 80%. In our setting the threshold's absolute value was now at 450MB before spilling to disk.

By default, the map output is uncompressed. However, compression can improve both performance and space usage [18, p. 209]. Therefore, we activate compression.

```
1 jobConf.setBoolean(MRJobConfig.MAP_OUTPUT_COMPRESS, true);
```

**List.** 5.7: Property mapreduce.map.output.compress

We chose Snappy as our compression codec. Snappy is patent-free and therefore included in the default distribution, at good performance[2].

```
1 jobConf.set(MRJobConfig.MAP_OUTPUT_COMPRESS_CODEC, "org.apache.hadoop.io.compress.
    SnappyCodec");
```

**List.** 5.8: Property mapreduce.map.output.compress.codec

[1] http://archive.cloudera.com/cdh4/cdh/4/hadoop/hadoop-project-dist/hadoop-common/ClusterSetup.html#Configuring_the_Hadoop_Daemons_in_Non-Secure_Mode

[2] https://code.google.com/p/snappy/

## 5.4. Reduce

Because our first map phase takes a long time to run, we don't want the reducers to start the *copy phase* too early. The reducers would otherwise take away useful RAM resources that we need for running the mappers. We set the value to 1.0 (instead of the default of 0.05).

```
1 jobConf.setFloat(MRJobConfig.COMPLETED_MAPS_FOR_REDUCE_SLOWSTART, 1.00f);
```

**List.** 5.9: Property mapreduce.job.reduce.slowstart.completedmaps

## 5.5. Benchmarking different configurations

To verify our settings, we proceed as follows. We use our intuited configuration as the baseline. Then, for each setting that is changed against the default, we run the entire pipeline generally three times: once with our intuited value, once with a 25 % smaller value, and once with a greater value. Then, if the value we intuited is not optimal, we set the better value to be the new baseline and iterate. We created a separate HAR file which contains only about 1 % of the repositories. The statistics of the different runs were collected by a Ruby script[3]. It queries information about the completed jobs on the "History Server" using the REST API[4]. Table 5.1 shows the average times taken (in seconds) for processing the 140 map tasks of the populate step, see section §3.2.1.

Our benchmark code can be seen in "HadoopConfigurationTest.java"[5].

This setup helps to efficiently identify relevant job configuration settings which speed up a MapReduce job: after studying a specific setting we regularly add another configuration test method to verify its effect.

[3] https://github.com/nes1983/cc/blob/master/scripts/hadoop/MRStat.rb

[4] http://archive.cloudera.com/cdh4/cdh/4/hadoop/hadoop-yarn/hadoop-yarn-site/HistoryServerRest.html#Job_API

[5] https://github.com/nes1983/cc/blob/master/src/ch/unibe/scg/cc/HadoopConfigurationTest.java

**Tab.** 5.1.: Time taken for 140 Map tasks (in seconds)

| **Name** | **Duration** | **∅Map** | **∅Reduce** | **∅Shuffle** | **∅Merge** |
|---|---|---|---|---|---|
| Spilling much lower | 918 | 30 | 65 | 663 | 674 |
| Spilling lower | 922 | 29 | 67 | 673 | 683 |
| IO Sort lower | 1019 | 30 | 58 | 773 | 784 |
| IO Sort higher | 1037 | 32 | 69 | 778 | 789 |
| SlowStart 50% | 1040 | 33 | 70 | 769 | 780 |
| Current CC configuration | 1042 | 31 | 84 | 776 | 788 |
| SlowStart 100% | 1044 | 35 | 63 | 10 | 23 |
| Reduce Mem2Mem, 3000M | 1046 | 31 | 64 | 785 | 794 |
| Reducers lower | 1051 | 31 | 71 | 875 | 888 |
| Reducers higher | 1054 | 32 | 45 | 560 | 567 |
| Speculative variation | 1055 | 32 | 59 | 797 | 808 |
| MapCompression off | 1057 | 31 | 67 | 795 | 806 |
| Shuffle Parallel Copies 10 | 1060 | 30 | 67 | 802 | 812 |
| Shuffle Parallel Copies 2 | 1061 | 31 | 69 | 801 | 811 |
| MemoryMB lower | 1063 | 36 | 88 | 870 | 882 |
| MemoryMB higher | 1066 | 29 | 51 | 594 | 602 |
| MapCompression w/ DefaultCodec | 1067 | 32 | 70 | 794 | 807 |
| Reduce Mem2Mem, 0M | 1069 | 33 | 62 | 802 | 812 |
| Spilling higher | 1071 | 32 | 70 | 802 | 813 |
| SlowStart 0% | 1072 | 32 | 63 | 804 | 816 |
| Reduce Mem2Mem, 1000M | 1079 | 33 | 64 | 809 | 818 |

# 6 HBase tuning



HBase seems to be a good match as an intermediate database during the clone detection process. One value we tune is `hbase.hregion.max.filesize`. It defines the threshold before table splitting gets initiated. The default of 256MB is too small for the bulk-loading during the populate stage because splitting may block the entire pipeline. Increasing the value to 1GB results in better performance in our case.

## 6.1. Compression

Just like Mapper output, HBase tables can be compressed. We chose Snappy[1] as the compression algorithm within our HBase table setup. It is designed to offer "very high speeds with a reasonable compression"[2]. We compared the compressed and uncompressed reducer output of the population stage. Without compression we generated 95 GB of data and with Snappy compression it shrunk to 19 GB which gives us a compression rate of 80%. This is interesting because hashes are generally not compressible. This implies that we have a lot of reoccurring hashes in our data. However, this example illustrates the need of a compression codec.

### 6.1.1. Scripting

To benchmark different table layouts, we need to truncate tables (or even recreate them from scratch) from time to time. We wrote a script to rebuild all tables from scratch; it can be run as `hbase shell scripts/hbase/hbaseTruncateTables`. However, after implementing the Cells [14] framework, there was no more need to do so because Cells creates temporary tables for intermediate output.

[1] Snappy: `https://code.google.com/p/snappy/`
[2] `http://www.infoq.com/news/2011/04/Snappy`

## 6.2. Pitfalls

When configuring HBase, one moment of unwariness can lead to unwanted effects. It happened to us that we had set the max region size `hbase.hregion.max.filesize` to a exceedingly small value (1048576). This number represents the file size in Bytes, while we assumed it was expressed in Megabytes. This had the effect that while bulk-loading HBase, the system constantly had to split the tables again and again. After some time there were so many tables that the system was completely busy just splitting up tables, so that the HBase Regionservers first produced timeouts and later on crashed. After investigating deeper in the logs and carefully checking the configuration values, we realized this silly mistake and adjusted to a greater value (1073741824 which corresponds to one Gigabyte).

### 6.2.1. Of strings and symbols

After we'd updated CDH4.1.2 to CDH4.2.0, we were unable to set the the HBase compression for new tables anymore. We posted our question on Stack Overflow[3]. We identified the problem on how JRuby compares strings and symbols, and were able to write a fix that lets HBase administration scripts work on JRuby.

[3] `http://www.stackoverflow.com/questions/15586111/cdh4-2-0-unable-to-set-hbase-compression`

# 7 Discussion



The time to detect clones across versions and across all Java-projects on Ohloh can be found in Table 7.1. The key ingredients to its performance are two-fold. First, we forbid any global reads across the cluster, leading to a *data-local* algorithm. Second, we employ *data reduction* by eliminating, as early as possible, all snippets that cannot partake in a clone, because they do not collide. This step alone eliminates 3/4 of the data.

## 7.1. Scale

While our current implementation is restricted to Java source code, we consider our mission to scale to all public source code to be accomplished. Java is, according to TIOBE[1], the second most popular programming language language, meaning that we're already covering a significant fraction of all open source code, on only three machines, in 24 hours of cluster time. Given that our pipeline computes locally, without global lookups, it is expected to scale up linearly to more machines.

[1] `http://www.tiobe.com/`, viewed in December 2013

**Tab.** 7.1.: Time taken in minutes, per mapper. Total time taken is 19 hours, 30 minutes

| Populate | Reverse Index | Filter | Rough clones | Fine clones |
|---|---|---|---|---|
| 427 | 60 | 25 | 207 | 451 |

## 7.2. Lessons learnt

Working with a small Hadoop cluster afforded us a chance to learn valuable lessons, although we are sure that they are well-known to the initiated.

### 7.2.1. Inadvertently forcing write-flushes

At first, for only 405 repositories, `populate` took over 60 minutes, which seemed fairly long. We later realized that this was caused by a small check to see whether or not a file has been written before. Alas, as we described earlier, reading while writing forces all write-buffers to be flushed, causing performance to degenerate. Removing the check and writing source trees regardless of whether or not they've been written before, which is safe, improved the time down to 2 minutes.

### 7.2.2. Leap second bug or how we won 1 second and lost a week

We were not aware of the Leap second which was added at at 23:59:60 UTC on June 30, 2012. Java processes used excessive amount of CPU. This led to very slow reaction times of Hadoop and HBase, restarted services multiple times. In the end, the only thing that effectively helped was a reboot of the machines. (see `https://bugzilla.mozilla.org/show_bug.cgi?id=769972`)

### 7.2.3. Pitfall: JRuby Bug

We experienced a strange failure while executing our download script[2].

```
1 Input/output error
2 ["org/jruby/RubyIO.java:1931:in ‘eof?’", "/home/vs/.rvm/rubies/jruby-1.7.3/lib/ruby
    /1.9/expect.rb:26:in ‘expect’", (...)
```

**List.** 7.1: JRuby console output

While we couldn't determine any possible reason for this behavior, switching from JRuby to standard Ruby behaved according to the documentation. Therefore, changing our default Ruby interpreter solved all issues.

[2]see `https://github.com/nes1983/cc/blob/master/scripts/ohloh/OhlohJavaRepoFetcher.rb`

# 8
# Related Work



There are two fundamentally different categories of clone detectors: clustering detectors, and table- or index-based detectors. Let us discuss the two categories in turn.

## 8.1. Index-based detectors

Index-based approaches, first suggested by Hummel *et al.* [7], 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. Our approach can be viewed as a refinement of theirs in two ways: first, Hummel's approach is not data-local, but requires random lookups across the cluster into a global table of methods during clone detection, making our detector scale much better.

Our technique of 'bad hashing' adds to all index-based approaches the ability to detect similar snippets that differ in only a few tokens. Without bad hashes, even small differences produce different hashes, and therefore remain undetected.

Our approach very much falls into the table-based category. We have previously built a clone detector using bad hashes to determine the amount of cloning that occurs at a function level [15]. To this, this thesis adds the ability to detect clones at the sub-method level, without sacrificing scalability.

Keivanloo *et al.* [9] 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, HBase. This enables us to run elaborate queries, like "how much cloning exists between *different* projects" within hours, even without the use of parallelization.

## 8.2. Clustering detectors

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 [8]. Livieri *et al.* [12] 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.* [17] show how simhashes 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.* [11] 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.

Chang and Mockus investigated source code reuse, with FreeBSD as a case study (57,128 files, 492,583 versions, 7.5 GB). They compared several techniques: identical file names, identical contents, trigrams, vector spaces, and abstract syntax trees [3]. They found that a large number of reused files were detectable based on file name only, and an equally large, but partially overlapping subset had identical contents.

Davies *et al.* [4] present a technique for determining the origin of a library based on the signature of the classes and methods inside. The signatures were hashed with SHA-1 during corpus indexing. The approach scales to the size of Maven (130,000 jar files, 150 GB), and was used to identify the version of the majority of jar files used by a commercial application.

## 8.3. Techniques

Our clone detector borrows the following techniques, which are helpful in detecting clones, but do not alone suffice.

Broder [2] uses runs of 4 consecutive words—called shingles—to compute the similarity of two documents. A subset of these runs is kept as the "sketch" of a document; comparing two documents boils down to counting how many shingles their sketches share. This gives a similarity metric: documents are considered similar if their distance is lower than a threshold. Broder detected clusters in 30,000,000 web documents totaling 150 GB. Dig *et al.* use shingles encoding to detect renamed methods across versions of software systems, in the context of refactoring detection [5].

The idea of using bad hashes for clone detection was proposed by Baxter *et al.* [1]. Their approach creates bad hashes for sub-trees of the ASTs of classes, and thus requires full parsing of the source code in question.

# 9 Conclusion

Our clone detector combines bad hashing, light-weight parsing using regular expressions, and MapReduce pipelines. We show that clone detection can be achieved without random reads across the network, in log-linear time, and therefore scales well. Our clone detector finds clones in all open source Java projects, across all versions.

# A Resources

## A.1. Repository

The whole source code of the clone detector and the scripts is available online under `https://github.com/nes1983/cc`.

# Bibliography


[1] Ira Baxter, Andrew Yahin, Leonardo Moura, Marcelo Sant' Anna, and Lorraine Bier. Clone detection using abstract syntax trees. In *Proceedings of the International Conference on Software Maintenance (ICSM 1998)*, pages 368–377. IEEE Computer Society, Washington, DC, USA, 1998. 28

[2] A. Z. Broder. On the resemblance and containment of documents. In *Proceedings of the Compression and Complexity of Sequences 1997*, SEQUENCES '97, pages 21–29, June 1997. 28

[3] Hung-Fu Chang and Audris Mockus. Evaluation of source code copy detection methods on freebsd. In *MSR '08: Proceedings of the 2008 international working conference on Mining software repositories*, pages 61–66, New York, NY, USA, 2008. ACM. 28

[4] Julius Davies, Daniel M. Germán, Michael W. Godfrey, and Abram Hindle. Software bertillonage: finding the provenance of an entity. In *MSR'11: Proceedings of the 8th International Working Conference on Mining Software Repositories*, pages 183–192, 2011. 28

[5] Danny Dig and Ralph Johnson. How do APIs evolve? a story of refactoring. *Journal of Software Maintenance and Evolution: Research and Practice (JSME)*, 18(2):83–107, April 2006. 28

[6] James R. Driscoll, Neil Sarnak, Daniel D. Sleator, and Robert E. Tarjan. Making data structures persistent. *Journal of Computer and System Sciences*, 38(1):86–124, February 1989. 8

[7] B. Hummel, E. Juergens, L. Heinemann, and M. Conradt. Index-based code clone detection: incremental, distributed, scalable. In *ICSM*, pages 1–9, 2010. 1, 27

[8] Toshihiro Kamiya, Shinji Kusumoto, and Katsuro Inoue. CCFinder: A multilinguistic token-based code clone detection system for large scale source code. *IEEE Transactions on Software Engineering*, 28(6):654–670, 2002. 2, 28

[9] Iman Keivanloo, Juergen Rilling, and Philippe Charland. Internet-scale real-time code clone search via multi-level indexing. In *WCRE*, pages 23–27, 2011. 27

[10] Rainer Koschke. Large-Scale Inter-System clone detection using suffix trees. In *Proceedings of the 2012 16th European Conference on Software Maintenance and Reengineering*, CSMR '12, pages 309–318, Washington, DC, USA, 2012. IEEE Computer Society. 3

[11] Jens Krinke, Nicolas Gold, Yue Jia, and David Binkley. Cloning and copying between gnome projects. In *MSR*, pages 98–101, 2010. 28

[12] Simone Livieri, Yoshiki Higo, Makoto Matsushita, and Katsuro Inoue. Very-large scale code clone analysis and visualization of open source programs using distributed ccfinder: D-ccfinder. In *ICSE*, pages 106–115, 2007. 28

[13] Niko Schwarz, Aaron Karper, and Oscar Nierstrasz. Efficient regular expressions that produce parse trees. Report, University of Bern, January 2014. submitted for publication. 10

[14] Niko Schwarz, Alexey Kolesnichenko, and Oscar Nierstrasz. Cells: Expressing parallel pipelines for local and cluster execution. Report, University of Bern, January 2014. submitted for publication. 8, 23

[15] Niko Schwarz, Mircea Lungu, and Romain Robbes. On how often code is cloned across repositories. In *Proceedings of the 2012 International Conference on Software Engineering*, ICSE 2012, pages 1289–1292, Piscataway, NJ, USA, 2012. IEEE Press. 3, 27

[16] O. Tange. GNU parallel - the Command-Line power tool. *;login: The USENIX Magazine*, 36(1):42–47, February 2011. 6

[17] Md. Sharif Uddin, Chanchal K. Roy, Kevin A. Schneider, and Abram Hindle. On the effectiveness of simhash for detecting near-miss clones in large scale software systems. In *WCRE*, pages 13–22, 2011. 28

[18] Tom White. *Hadoop: The Definitive Guide*. O'Reilly Media, third edition edition, May 2012. 20